EBook Gratis

APRENDIZAJE Haskell Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#haskell

Tabla de contenido

Acerca de ... 1

Capítulo 1: Empezando con Haskell Language ... 2

Observaciones ... 2

caracteristicas: ... 2

Versiones ... 2

Examples ... 3

¡Hola Mundo! ... 3

Explicación: ... 4

Factorial ... 5

Variación 1 ... 5

Variación 2 ... 6

Fibonacci, utilizando la evaluación perezosa ... 6

Empezando ... 7

REPL en linea ... 7

GHC (i) ... 7

Herramientas mas avanzadas ... 9

Primes ... 10

Por debajo de 100 ... 10

Ilimitado ... 10

Tradicional ... 10

División de prueba óptima ... 10

Transicional ... 10

El código más corto ... 11

Declarar valores ... 11

Capítulo 2: Agujeros mecanografiados ... 12

Observaciones ... 12

Examples ... 12

Sintaxis de los agujeros mecanografiados. ... 12

Controlando el comportamiento de los agujeros mecanografiados. ... 12

Semántica de agujeros mecanografiados. ... 13

Usando agujeros escritos para definir una instancia de clase 13
Capítulo 3: Analizando HTML con lentes etiquetadas y lentes 16
Examples 16
Extrae el contenido del texto de un div con un id particular 16
Filtrar elementos del árbol 16
Capítulo 4: Apilar 18
Examples 18
Instalación de la pila 18
Creando un proyecto simple 18
Estructura 18
Estructura de archivos 18
Ejecutando el programa 18
Paquetes de apilamiento y cambio de la versión LTS (resolución) 19
Añadiendo lentes a un proyecto. 19
Construir y ejecutar un proyecto de pila 19
Instalación de pila 20
Perfilando con Stack 20
Visualización de dependencias 20
Capítulo 5: Aritmética 22
Introducción 22
Observaciones 22
La jerarquía de clases de tipo numérico. 22
Examples 24
Ejemplos basicos 24
`No se pudo deducir (Fraccional Int) ...` 24
Ejemplos de funciones 25
Capítulo 6: Atravesable 26
Introducción 26
Examples 26
Funcionalizador de funciones y plegable para una estructura transversal 26
Una instancia de Traversable para un árbol binario 27

Atravesando una estructura en reversa ... 28
Definición de Traversable ... 28
Transformación de una estructura transitable con la ayuda de un parámetro de acumulación ... 29
Estructuras transitables como formas con contenidos ... 30
Transponer una lista de listas ... 30
Capítulo 7: Attoparsec ... 32
Introducción ... 32
Parámetros ... 32
Examples ... 32
Combinadores ... 32
Mapa de bits - Análisis de datos binarios ... 33
Capítulo 8: Bases de datos ... 35
Examples ... 35
Postgres ... 35
Sustitución de parámetros ... 35
Ejecución de inserciones o actualizaciones ... 35
Capítulo 9: Bifunctor ... 36
Sintaxis ... 36
Observaciones ... 36
Examples ... 36
Instancias comunes de Bifunctor ... 36
Tuplas de dos elementos ... 36
Either ... 36
primero y segundo ... 37
Definición de Bifunctor ... 37
Capítulo 10: Cábala ... 38
Sintaxis ... 38
Examples ... 39
Instalar paquetes ... 39
Trabajando con cajas de arena ... 39
Capítulo 11: Categoria teoria ... 41

Examples 41
La teoría de la categoría como sistema de organización de la abstracción 41
Un ejemplo 41
Un indicio de la teoría de la categoría. 41
Definición de una categoría 42
Haskell tipifica como categoría 43
Definición de la categoría 43
Isomorfismos 43
Functores 44
Mónadas 44
Producto de tipos en Hask 44
Productos categoricos 45
Productos en Hask 45
Unicidad hasta el isomorfismo 45
Singularidad de la descomposición 46
Coproducto de tipos en Hask 46
Intuición 46
Coproductos categóricos 46
Coproductos en Hask 47
Haskell Applicative en términos de la teoría de la categoría 47
Capítulo 12: Clases de tipo 49
Introducción 49
Observaciones 49
Examples 49
Tal vez y la clase Functor 49
Herencia de clase de tipo: Ord clase de tipo 50
Ecuación 51
Metodos requeridos 51
Define 51
Superclases directas 51
Subclases notables 52

Ord ... 52
Metodos requeridos ... 52
Define ... 52
Superclases directas ... 52
Monoide ... 52
Metodos requeridos ... 53
Superclases directas ... 53
Num ... 53
Los métodos ... 53
Capítulo 13: Clasificación de los algoritmos ... 56
Examples ... 56
Tipo de inserción ... 56
Combinar clasificación ... 56
Ordenación rápida ... 57
Ordenamiento de burbuja ... 57
Orden de permutación ... 57
Selección de selección ... 57
Capítulo 14: Composición de funciones ... 59
Observaciones ... 59
Examples ... 60
Composición de derecha a izquierda ... 60
Composición de izquierda a derecha ... 60
Composición con función binaria ... 60
Capítulo 15: Comprobación rápida ... 62
Examples ... 62
Declarar una propiedad ... 62
Comprobando una sola propiedad ... 62
Comprobando todas las propiedades en un archivo ... 62
Generando datos al azar para tipos personalizados ... 63
Usando la implicación (==>) para verificar las propiedades con condiciones previas ... 63
Limitar el tamaño de los datos de prueba ... 63

Capítulo 16: Concurrencia ... 65
Observaciones ... 65
Examples ... 65
Hilos de desove con `forkIO` ... 65
Comunicando entre hilos con `MVar` ... 65
Bloques atómicos con software de memoria transaccional ... 66
atomically :: STM a -> IO a ... 67
readTVar :: TVar a -> STM a ... 67
writeTVar :: TVar a -> a -> STM () ... 67
Capítulo 17: Contenedores - Data.Map ... 68
Examples ... 68
Construyendo ... 68
Comprobando si está vacío ... 68
Encontrar valores ... 68
Insertando Elementos ... 69
Borrando elementos ... 69
Importando el Módulo ... 69
Instancia de monoide ... 69
Capítulo 18: Creación de tipos de datos personalizados ... 71
Examples ... 71
Creando un tipo de datos simple ... 71
Creando variables de nuestro tipo personalizado. ... 71
Creación de un tipo de datos con parámetros de constructor de valor. ... 71
Creando variables de nuestro tipo personalizado. ... 72
Creación de un tipo de datos con parámetros de tipo ... 72
Creando variables de nuestro tipo personalizado. ... 72
Tipo de datos personalizado con parámetros de registro ... 72
Capítulo 19: Data.Aeson - JSON en Haskell ... 74
Examples ... 74
Codificación y decodificación inteligentes usando genéricos ... 74
Una forma rápida de generar un Data.Aeson.Value ... 75
Campos opcionales ... 75

Capítulo 20: Data.Text ... 76
Observaciones ... 76
Examples ... 76
Literales de texto ... 76
Eliminar espacios en blanco ... 76
Dividir valores de texto ... 77
Codificación y decodificación de texto ... 77
Comprobando si un texto es una subcadena de otro texto ... 78
Texto de indexación ... 78
Capítulo 21: Declaraciones de fijeza ... 80
Sintaxis ... 80
Parámetros ... 80
Observaciones ... 80
Examples ... 81
Asociatividad ... 81
Precedencia vinculante ... 81
Observaciones ... 82
Declaraciones de ejemplo ... 82
Capítulo 22: Desarrollo web ... 83
Examples ... 83
Servidor ... 83
Yesod ... 84
Capítulo 23: Esquemas de recursion ... 86
Observaciones ... 86
Examples ... 86
Puntos fijos ... 86
Doblando una estructura de una capa a la vez ... 87
Desplegar una estructura de una capa a la vez ... 87
Despliegue y luego plegado, fusionado ... 87
Recursion primitiva ... 88
Corecursion primitiva ... 88
Capítulo 24: Explotación florestal ... 89

Introducción ... 89
Examples ... 89
Registro con hslogger ... 89
Capítulo 25: Extensiones de lenguaje GHC comunes ... 90
Observaciones ... 90
Examples ... 90
MultiParamTypeClasses ... 90
FlexibleInstances ... 90
SobrecargadoStrings ... 91
TupleSections ... 91
N-tuplas ... 91
Cartografía ... 92
UnicodeSyntax ... 92
Literales binarios ... 93
Cuantificación existencial ... 93
LambdaCase ... 94
RankNTypes ... 95
Listas sobrecargadas ... 96
Dependencias funcionales ... 97
GADTs ... 97
ScopedTypeVariables ... 98
Sinónimo de patrones ... 98
RecordWildCards ... 100
Capítulo 26: Fecha y hora ... 101
Sintaxis ... 101
Observaciones ... 101
Examples ... 101
Encontrar la fecha de hoy ... 101
Sumando, restando y comparando días ... 101
Capítulo 27: Flechas ... 103
Examples ... 103

Composiciones de funciones con múltiples canales....103
Capítulo 28: Función de sintaxis de llamada....105
Introducción....105
Observaciones....105
Examples....105
Paréntesis en una función básica llamada....105
Paréntesis en llamadas a funciones incrustadas....105
Solicitud parcial - Parte 1....106
Aplicación parcial - Parte 2....106
Capítulo 29: Funciones de orden superior....108
Observaciones....108
Examples....108
Conceptos básicos de las funciones de orden superior....108
Expresiones lambda....109
Zurra....110
Capítulo 30: Functor....111
Introducción....111
Observaciones....111
Identidad....111
Composición....111
Examples....111
Instancias comunes de Functor....111
Tal vez....111
Liza....112
Funciones....113
Definición de clase de Functor y Leyes....114
Reemplazo de todos los elementos de un Functor con un solo valor....114
Funtores polinomiales....114
El functor de la identidad....114
El functor constante....115
Productos funcionales....115

Functor coproductos ... 115
Composición funcional ... 116
Funtores polinómicos para la programación genérica. ... 116
Functors en la teoría de categorías ... 117
Functor de derivación ... 118
Capítulo 31: Functor Aplicativo ... 119
Introducción ... 119
Observaciones ... 119
Definición ... 119
Examples ... 119
Definición alternativa ... 119
Ejemplos comunes de aplicativo ... 120
Tal vez ... 120
Liza ... 120
Flujos infinitos y listas zip ... 120
Funciones ... 121
Capítulo 32: GHCJS ... 123
Introducción ... 123
Examples ... 123
Ejecutando "¡Hola mundo!" con Node.js ... 123
Capítulo 33: Google Protocol Buffers ... 124
Observaciones ... 124
Examples ... 124
Creando, construyendo y usando un simple archivo .proto ... 124
Capítulo 34: Gráficos con brillo ... 126
Examples ... 126
Instalar Gloss ... 126
Consiguiendo algo en la pantalla ... 126
Capítulo 35: Gtk3 ... 128
Sintaxis ... 128
Observaciones ... 128

Examples ... 128
Hola mundo en gtk ... 128
Capítulo 36: Interfaz de función extranjera ... 130
Sintaxis ... 130
Observaciones ... 130
Examples ... 130
Llamando C desde Haskell ... 130
Pasar funciones de Haskell como devoluciones de llamada a código C ... 131
Capítulo 37: IO ... 133
Examples ... 133
Leyendo todos los contenidos de entrada estándar en una cadena ... 133
Leyendo una línea de entrada estándar ... 133
Analizar y construir un objeto desde la entrada estándar ... 133
Leyendo desde los manejadores de archivos ... 134
Comprobación de condiciones de fin de archivo ... 135
Leyendo palabras de un archivo completo ... 135
IO define la acción `main` de su programa ... 136
Papel y propósito de IO ... 137
Manipulando valores IO ... 137
Semántica IO ... 138
Perezoso IO ... 139
IO y do la notación ... 139
Obtener la 'a' "de" 'IO a' ... 140
Escribiendo al stdout ... 140
putChar :: Char -> IO () - escribe un char en stdout ... 140
putStr :: String -> IO () - escribe un String en stdout ... 141
putStrLn :: String -> IO () - escribe un String en stdout y agrega una nueva línea ... 141
print :: Show a => a -> IO () - escribe a instancia de Show a stdout ... 141
Leyendo de `stdin` ... 141
getChar :: IO Char - lee un Char de stdin ... 141
getLine :: IO String - lee un String desde stdin , sin una nueva línea de caracteres ... 142

read :: Read a => String -> a - convierte un String a un valor 142
Capítulo 38: Lector / Lector 143
Introducción 143
Examples 143
Simple demostración 143
Capítulo 39: Lente 145
Introducción 145
Observaciones 145
¿Qué es una lente? 145
Enfoque 145
Otras ópticas 145
Composición 146
En haskell 146
Examples 147
Manipulación de tuplas con lente 147
Lentes para discos 147
Registro simple 147
Gestión de registros con nombres de campos repetidos 147
Lentes de estado 148
Deshacerse de & cadenas 148
Código imperativo con estado estructurado. 148
Escribiendo una lente sin plantilla Haskell 149
Lente y prisma 149
Travesías 150
Lentes componen 151
Lentes con clase 151
Campos con makeFields 151
Capítulo 40: Lista de Comprensiones 154
Examples 154
Lista de comprensión básica 154
Patrones en Expresiones de Generador 154

Guardias 155
Generadores anidados 155
Comprensiones paralelas 155
Fijaciones locales 156
Hacer notación 156
Capítulo 41: Literales sobrecargados 157
Observaciones 157
Literales enteros 157
Literales fraccionales 157
Literales de cuerda 157
Lista de literales 157
Examples 158
Numero entero 158
El tipo del literal 158
Elegir un tipo concreto con anotaciones 158
Numeral flotante 158
El tipo del literal 158
Elegir un tipo concreto con anotaciones 158
Instrumentos de cuerda 159
El tipo del literal 159
Usando cadenas literales 159
Lista de literales 160
Capítulo 42: Liza 161
Sintaxis 161
Observaciones 161
Examples 162
Lista de literales 162
Concatenación de listas 162
Conceptos básicos de la lista 162
Procesando listas 163
Accediendo a elementos en listas 164
Gamas 164

Funciones básicas en listas ... 165
pliegue ... 166
plegar ... 166
Transformando con `map` ... 166
Filtrado con `filter` ... 167
Listas de cierre y descompresión ... 167
Capítulo 43: Los funtores comunes como base de los comonads cofree. ... 169
Examples ... 169
Cofree Empty ~~ Empty ... 169
Cofree (Const c) ~~ escritor c ... 169
Cofree Identity ~~ Stream ... 169
Cofree Maybe ~~ NonEmpty ... 170
Cofree (Writer w) ~~ WriterT w Stream ... 170
Cofree (Either e) ~~ NonEmptyT (Writer e) ... 170
Cofree (Lector x) ~~ Moore x ... 171
Capítulo 44: Mejoramiento ... 172
Examples ... 172
Compilando su programa para perfilar ... 172
Centros de costo ... 173
Capítulo 45: Módulos ... 174
Sintaxis ... 174
Observaciones ... 174
Examples ... 174
Definiendo tu propio módulo ... 174
Constructores exportadores ... 175
Importación de miembros específicos de un módulo ... 175
Ocultar Importaciones ... 175
Importaciones Calificadas ... 176
Nombres de módulos jerárquicos ... 176
Capítulo 46: Mónada del estado ... 178
Introducción ... 178
Observaciones ... 178

Examples ... 178
Numerar los nodos de un árbol con un contador ... 178
El largo camino ... 178
Refactorización ... 179
Dividir el contador en una acción postIncrement ... 179
Divide la caminata del árbol en una función de orden superior ... 180
Usa la clase Traversable ... 180
Deshacerse de la Traversable desplazamiento de Traversable ... 180
Capítulo 47: Mónadas ... 182
Introducción ... 182
Examples ... 182
La mónada Tal vez ... 182
IO mónada ... 184
Lista Mónada ... 185
La mónada como subclase de aplicativo ... 186
No hay forma general de extraer valor de un cálculo monádico ... 186
hacer notación ... 187
Definición de mónada ... 187
Capítulo 48: Mónadas comunes como mónadas libres. ... 189
Examples ... 189
Libre Vacío ~~ Identidad ... 189
Identidad libre ~~ (Nat,) ~~ escritor Nat ... 189
Libre Tal vez ~~ MaybeT (Escritor Nat) ... 190
Libre (escritor w) ~~ escritor [w] ... 190
Libre (Const c) ~~ O bien c ... 190
Gratis (Reader x) ~~ Reader (Stream x) ... 191
Capítulo 49: Mónadas Libres ... 192
Examples ... 192
Las mónadas libres dividen los cálculos monádicos en estructuras de datos e intérpretes ... 192
Las mónadas libres son como puntos fijos ... 193
¿Cómo funcionan foldFree y iterM? ... 193
La mónada Freer ... 194

Capítulo 50: Monoide 197
Examples 197
Un ejemplar de monoide para listas 197
Contraer una lista de Monoids en un solo valor 197
Monoides Numéricos 197
Una instancia de monoide para () 198
Capítulo 51: Operadores de infijo 199
Observaciones 199
Examples 199
Preludio 199
Lógico 199
Operadores aritméticos 199
Liza 200
Flujo de control 200
Operadores personalizados 200
Encontrar información sobre operadores de infijo 201
Capítulo 52: Papel 202
Introducción 202
Observaciones 202
Examples 202
Papel nominal 202
Papel representativo 202
Papel fantasma 203
Capítulo 53: Paralelismo 204
Parámetros 204
Observaciones 204
Examples 205
La Mónada Eval 205
rpar 205
rseq 206
Capítulo 54: Plantilla Haskell & QuasiQuotes 207

Observaciones ... 207
¿Qué es la plantilla Haskell? ... 207
¿Qué son las etapas? (O, ¿cuál es la restricción de la etapa?) ... 207
¿El uso de Template Haskell causa errores no relacionados con el alcance de identificadore ... 207
Examples ... 208
El tipo q ... 208
Un curry n-arity ... 209
Sintaxis de plantilla Haskell y cuasiquotes ... 210
Empalmes ... 210
Citas de expresión (nota: no una cotización) ... 211
Empalmes mecanografiados y citas ... 211
Cuasi Citas ... 212
Los nombres ... 212
Capítulo 55: Plátano reactivo ... 213
Examples ... 213
Inyectando eventos externos en la biblioteca ... 213
Tipo de evento ... 213
Tipo de comportamiento ... 214
Actuating EventNetworks ... 215
Capítulo 56: Plegable ... 216
Introducción ... 216
Observaciones ... 216
Examples ... 216
Contando los elementos de una estructura plegable ... 216
Doblando una estructura al revés ... 216
Una instancia de Plegable para un árbol binario ... 217
Aplanando una estructura plegable en una lista ... 218
Realización de un efecto secundario para cada elemento de una estructura plegable ... 218
Aplanando una estructura plegable en un monoide ... 219
Definición de plegable ... 220
Comprobando si una estructura plegable está vacía ... 220

Capítulo 57: Polimorfismo de rango arbitrario con RankNTypes ... 222
Introducción ... 222
Sintaxis ... 222
Examples ... 222
RankNTypes ... 222
Capítulo 58: Probando con Tasty ... 223
Examples ... 223
SmallCheck, QuickCheck y HUnit ... 223
Capítulo 59: Profesor ... 224
Introducción ... 224
Sintaxis ... 224
Observaciones ... 224
Examples ... 225
(->) Profesor ... 225
Capítulo 60: Proxies ... 226
Examples ... 226
Usando Proxy ... 226
El lenguaje "proxy polimórfico" ... 226
Proxy es como () ... 227
Capítulo 61: Reglas de reescritura (GHC) ... 228
Examples ... 228
Usando reglas de reescritura en funciones sobrecargadas ... 228
Capítulo 62: Rigor ... 229
Examples ... 229
Patrones de explosión ... 229
Formas normales ... 229
Forma normal reducida ... 229
Forma normal de la cabeza débil ... 229
Patrones perezosos ... 230
Campos estrictos ... 231
Capítulo 63: Sintaxis de grabación ... 233

Examples ... 233
Sintaxis basica ... 233
Copiar registros al cambiar los valores de campo ... 234
Graba con newtype ... 234
RecordWildCards ... 235
Definición de un tipo de datos con etiquetas de campo ... 235
La coincidencia de patrones ... 236
Coincidencia de patrones con NamedFieldPuns ... 236
Patrón a juego con RecordWildcards ... 236
Actualizaciones de registro ... 236
Capítulo 64: Sintaxis en funciones ... 237
Examples ... 237
Guardias ... 237
La coincidencia de patrones ... 237
Usando donde y guardias ... 238
Capítulo 65: Solicitud parcial ... 240
Observaciones ... 240
Examples ... 240
Función de adición parcialmente aplicada ... 240
Devolviendo una Función Parcialmente Aplicada ... 241
Secciones ... 241
Una nota sobre la resta ... 241
Capítulo 66: Streaming IO ... 243
Examples ... 243
Streaming IO ... 243
Capítulo 67: Tipo algebra ... 244
Examples ... 244
Números naturales en tipo álgebra ... 244
Tipos de union finita ... 244
Unicidad hasta el isomorfismo ... 244
Uno y cero ... 245

Suma y multiplicación ... 245
Reglas de suma y multiplicación. ... 246
Tipos recursivos ... 246
Liza ... 246
Arboles ... 247
Derivados ... 247
Funciones ... 247
Capítulo 68: Tipo de solicitud ... 248
Introducción ... 248
Examples ... 248
Evitar anotaciones de tipo ... 248
Escriba aplicaciones en otros idiomas ... 249
Orden de parametros ... 249
Interacción con tipos ambiguos ... 250
Capítulo 69: Tipo Familias ... 252
Examples ... 252
Tipo de familias de sinónimos ... 252
Familias de sinónimos de tipo cerrado ... 252
Abrir familias de sinónimos de tipo ... 252
Sinónimos asociados a clases ... 252
Familias de tipos de datos ... 253
Familias de datos independientes ... 253
Familias de datos asociados ... 254
La inyectividad ... 254
Capítulo 70: Tipos de datos algebraicos generalizados ... 255
Examples ... 255
Uso básico ... 255
Capítulo 71: Tipos fantasma ... 256
Examples ... 256
Caso de uso para tipos fantasmas: Monedas ... 256
Capítulo 72: Transformadores de mónada ... 257

Examples ... 257
Un contador monádico ... 257
Añadiendo un entorno ... 258
Los requisitos cambiaron: necesitamos registro! ... 258
Haciendo todo de una vez ... 259
Capítulo 73: Tubería ... 261
Observaciones ... 261
Examples ... 261
Productores ... 261
Los consumidores ... 261
Tubería ... 262
Corriendo tuberías con runEffect ... 262
Tubos de conexión ... 262
El transformador proxy mónada ... 263
Combinación de tuberías y comunicación en red ... 263
Capítulo 74: Tuplas (pares, triples, ...) ... 266
Observaciones ... 266
Examples ... 266
Construir valores de tupla ... 266
Escribe tipos de tuplas ... 266
Patrón de coincidencia en las tuplas ... 267
Extraer componentes de la tupla ... 267
Aplique una función binaria a una tupla (uncurrying) ... 268
Aplicar una función de tupla a dos argumentos (currying) ... 268
Intercambiar pares de componentes ... 268
Rigidez de emparejar una tupla ... 269
Capítulo 75: Usando GHCi ... 270
Observaciones ... 270
Examples ... 270
Iniciando GHCi ... 270
Cambiar el indicador predeterminado de GHCi ... 270
El archivo de configuración de GHCi ... 270

Cargando un archivo ... 270
Dejar de fumar GHCi ... 271
Recargando un archivo ya cargado ... 271
Puntos de ruptura con GHCi ... 271
Declaraciones multilínea ... 272
Capítulo 76: Vectores ... 273
Observaciones ... 273
Examples ... 273
El módulo Data.Vector ... 273
Filtrando un vector ... 274
Mapeo (`map`) y reducción (` fold`) de un vector ... 274
Trabajando en múltiples vectores ... 274
Capítulo 77: XML ... 275
Introducción ... 275
Examples ... 275
Codificación de un registro utilizando la biblioteca `xml` ... 275
Capítulo 78: zipWithM ... 276
Introducción ... 276
Sintaxis ... 276
Examples ... 276
Cálculos de precios de venta ... 276
Creditos ... 277

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: haskell-language

It is an unofficial and free Haskell Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Haskell Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Haskell Language

Observaciones

Haskell es un lenguaje de programación avanzado puramente funcional.

caracteristicas:

- **Escrito estáticamente:** cada expresión en Haskell tiene un tipo que se determina en el momento de la compilación. La verificación de tipo estático es el proceso de verificación de la seguridad de tipo de un programa basado en el análisis del texto de un programa (código fuente). Si un programa pasa un verificador de tipo estático, entonces se garantiza que el programa satisfaga algún conjunto de propiedades de seguridad de tipo para todas las entradas posibles.
- **Puramente funcional** : cada función en Haskell es una función en el sentido matemático. No hay declaraciones ni instrucciones, solo expresiones que no pueden mutar variables (locales o globales) ni acceder a estados como el tiempo o números aleatorios.
- **Concurrente:** su compilador insignia, GHC, viene con un recolector de basura paralelo de alto rendimiento y una biblioteca de concurrencia liviana que contiene una serie de primitivas y abstracciones de concurrencia útiles.
- **Evaluación perezosa: las** funciones no evalúan sus argumentos. Retrasa la evaluación de una expresión hasta que se necesita su valor.
- **Uso general:** Haskell está diseñado para ser utilizado en todos los contextos y entornos.
- **Paquetes: la** contribución de código abierto a Haskell es muy activa con una amplia gama de paquetes disponibles en los servidores de paquetes públicos.

El último estándar de Haskell es Haskell 2010. A partir de mayo de 2016, un grupo está trabajando en la próxima versión, Haskell 2020.

La documentación oficial de Haskell es también un recurso completo y útil. Excelente lugar para encontrar libros, cursos, tutoriales, manuales, guías, etc.

Versiones

Versión	Fecha de lanzamiento
Haskell 2010	2012-07-10
Haskell 98	2002-12-01

Examples

¡Hola Mundo!

Un básico "¡Hola mundo!" El programa en Haskell se puede expresar de manera concisa en solo una o dos líneas:

```
main :: IO ()
main = putStrLn "Hello, World!"
```

La primera línea es una anotación de tipo opcional, que indica que `main` es un valor de tipo `IO ()` , que representa una acción de E / S que "calcula" un valor de tipo `()` (lea "unidad"; la tupla vacía no transmite información) además de realizar algunos efectos secundarios en el mundo exterior (aquí, imprimir una cadena en el terminal). Esta anotación de tipo se suele omitir para `main` porque es su *único* tipo posible.

Coloca esto en un archivo `helloworld.hs` y compílalo usando un compilador de Haskell, como GHC:

```
ghc helloworld.hs
```

La ejecución del archivo compilado dará como resultado la salida `"Hello, World!"` siendo impreso a la pantalla:

```
./helloworld
Hello, World!
```

Alternativamente, `runhaskell` o `runghc` hacen posible ejecutar el programa en modo interpretado sin tener que compilarlo:

```
runhaskell helloworld.hs
```

El REPL interactivo también se puede utilizar en lugar de compilar. Se entrega con la mayoría de los entornos de Haskell, como `ghci` que viene con el compilador de GHC:

```
ghci> putStrLn "Hello World!"
Hello, World!
ghci>
```

Alternativamente, cargue scripts en ghci desde un archivo usando `load` (o `:l`):

```
ghci> :load helloworld
```

`:reload` (o `:r`) `:reload` todo en ghci:

```
Prelude> :l helloworld.hs
[1 of 1] Compiling Main             ( helloworld.hs, interpreted )
```

```
<some time later after some edits>

*Main> :r
Ok, modules loaded: Main.
```

Explicación:

Esta primera línea es una firma de tipo, declarando el tipo de `main` :

```
main :: IO ()
```

Los valores de tipo `IO ()` describen acciones que pueden interactuar con el mundo exterior.

Debido a que Haskell tiene un sistema de tipo Hindley-Milner completo que permite la inferencia de tipos automática, las firmas de tipos son técnicamente opcionales: si simplemente omite `main :: IO ()` , el compilador podrá inferir el tipo por sí mismo. Analizando la *definición* de `main` . Sin embargo, se considera un estilo muy malo no escribir firmas de tipo para definiciones de nivel superior. Las razones incluyen:

- Las firmas de tipos en Haskell son una pieza de documentación muy útil porque el sistema de tipos es tan expresivo que a menudo se puede ver qué tipo de cosa es buena para una función simplemente observando su tipo. Se puede acceder cómodamente a esta "documentación" con herramientas como GHCi. ¡Y a diferencia de la documentación normal, el verificador de tipos del compilador se asegurará de que realmente coincida con la definición de la función!

- Las firmas de tipos *mantienen los errores locales* . Si comete un error en una definición sin proporcionar su tipo de firma, es posible que el compilador no informe un error de inmediato, sino que simplemente infiera un tipo sin sentido para el mismo, con el que realmente verifica. Luego puede obtener un mensaje de error críptico cuando *use* ese valor. Con una firma, el compilador es muy bueno para detectar errores justo donde ocurren.

Esta segunda línea hace el trabajo real:

```
main = putStrLn "Hello, World!"
```

Si proviene de un lenguaje imperativo, puede ser útil tener en cuenta que esta definición también se puede escribir como:

```
main = do {
   putStrLn "Hello, World!" ;
   return ()
   }
```

O de manera equivalente (Haskell tiene un análisis basado en el diseño; pero *tenga cuidado al mezclar las pestañas y los espacios de manera inconsistente,* lo que confundirá este mecanismo):

```
main = do
    putStrLn "Hello, World!"
    return ()
```

Cada línea en un bloque `do` representa algún *cálculo* monádico (aquí, E / S), de modo que todo el bloque `do` representa la acción general que comprende estos subpasos combinándolos de una manera específica a la mónada dada (para I / O esto significa simplemente ejecutándolos uno tras otro).

La sintaxis `do` es en sí misma un azúcar sintáctico para las mónadas, como `IO` aquí, y el `return` es una acción no operativa que produce su argumento sin realizar efectos secundarios ni cálculos adicionales que puedan formar parte de una definición de mónada particular.

Lo anterior es lo mismo que definir `main = putStrLn "Hello, World!"` , porque el valor `putStrLn "Hello, World!"` Ya tiene el tipo `IO ()` . Visto como una "declaración", `putStrLn "Hello, World!"` puede verse como un programa completo, y usted simplemente define `main` para referirse a este programa.

Puedes consultar la firma de `putStrLn` línea :

```
putStrLn :: String -> IO ()
-- thus,
putStrLn (v :: String) :: IO ()
```

`putStrLn` es una función que toma una cadena como argumento y genera una acción de E / S (es decir, un valor que representa un programa que el tiempo de ejecución puede ejecutar). El tiempo de ejecución siempre ejecuta la acción denominada `main` , por lo que simplemente debemos definirla como igual a `putStrLn "Hello, World!"` .

Factorial

La función factorial es un Haskell "Hello World!" (y para la programación funcional en general) en el sentido de que demuestra sucintamente los principios básicos del lenguaje.

Variación 1

```
fac :: (Integral a) => a -> a
fac n = product [1..n]
```

Demo en vivo

- `Integral` es la clase de tipos de números integrales. Los ejemplos incluyen `Int` y `Integer` .
- `(Integral a) =>` coloca una restricción en el tipo `a` para estar en dicha clase
- `fac :: a -> a` dice que `fac` es una función que toma una `a` y devuelve una `a`
- `product` es una función que acumula todos los números en una lista al multiplicarlos.
- `[1..n]` es una notación especial que se aplica a `enumFromTo 1 n` , y es el rango de números `1 ≤ x ≤ n` .

Variación 2

```
fac :: (Integral a) => a -> a
fac 0 = 1
fac n = n * fac (n - 1)
```

Demo en vivo

Esta variación utiliza la coincidencia de patrones para dividir la definición de función en casos separados. La primera definición se invoca si el argumento es `0` (a veces se denomina condición de detención) y la segunda definición de lo contrario (el orden de las definiciones es significativo) También ejemplifica la recursión como `fac` refiere a sí mismo.

Vale la pena señalar que, debido a las reglas de reescritura, ambas versiones de `fac` se compilarán con un código de máquina idéntico al usar GHC con las optimizaciones activadas. Entonces, en términos de eficiencia, los dos serían equivalentes.

Fibonacci, utilizando la evaluación perezosa

Evaluación perezosa significa que Haskell evaluará solo los elementos de la lista cuyos valores son necesarios.

La definición recursiva básica es:

```
f (0)  <-  0
f (1)  <-  1
f (n)  <-  f (n-1) + f (n-2)
```

Si se evalúa directamente, será *muy* lento. Pero, imagina que tenemos una lista que registra todos los resultados,

```
fibs !! n  <-  f (n)
```

Entonces

```
                  ┌──────┐   ┌──────┐   ┌──────┐
                  │ f(0) │   │ f(1) │   │ f(2) │
fibs  ->  0 : 1 : │  +   │ : │  +   │ : │  +   │ :  .....
                  │ f(1) │   │ f(2) │   │ f(3) │
                  └──────┘   └──────┘   └──────┘

                  ┌──────────────────────────────────┐
                  │ f(0)   :   f(1)   :   f(2)   :  ..... │
                  └──────────────────────────────────┘
      ->  0 : 1 :              +
                  ┌──────────────────────────────────┐
                  │ f(1)   :   f(2)   :   f(3)   :  ..... │
                  └──────────────────────────────────┘
```

Esto se codifica como:

```
fibn n = fibs !! n
    where
    fibs = 0 : 1 : map f [2..]
    f n = fibs !! (n-1) + fibs !! (n-2)
```

O incluso como

```
GHCi> let fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
GHCi> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

`zipWith` una lista aplicando una función binaria dada a los elementos correspondientes de las dos listas que se le asignaron, por lo que `zipWith (+) [x1, x2, ...] [y1, y2, ...]` es igual a `[x1 + y1, x2 + y2, ...]` .

Otra forma de escribir `fibs` es con la función `scanl` :

```
GHCi> let fibs = 0 : scanl (+) 1 fibs
GHCi> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

`scanl` construye la lista de resultados parciales que `foldl` produciría, trabajando de izquierda a derecha a lo largo de la lista de entrada. Es decir, `scanl f z0 [x1, x2, ...]` es igual a `[z0, z1, z2, ...] where z1 = f z0 x1; z2 = f z1 x2; ...`

Gracias a la evaluación perezosa, ambas funciones definen listas infinitas sin calcularlas por completo. Es decir, podemos escribir una función `fib` , recuperando el elemento nth de la secuencia de Fibonacci ilimitada:

```
GHCi> let fib n = fibs !! n  -- (!!) being the list subscript operator
-- or in point-free style:
GHCi> let fib = (fibs !!)
GHCi> fib 9
34
```

Empezando

REPL en linea

La forma más fácil de comenzar a escribir Haskell es probablemente ir al sitio web de Haskell o Probar Haskell y usar el REPL (read-eval-print-loop) en línea en la página de inicio. El REPL en línea admite la mayoría de las funciones básicas e incluso algunas IO. También hay un tutorial básico disponible que puede iniciarse escribiendo la `help` comando. Una herramienta ideal para comenzar a aprender los conceptos básicos de Haskell y probar algunas cosas.

GHC (i)

Para los programadores que están listos para involucrarse un poco más, existe *GHCi* , un entorno interactivo que viene con el *compilador Haskell de Glorious / Glasgow* . El *GHC* se puede instalar por separado, pero eso es solo un compilador. Para poder instalar nuevas bibliotecas, también se deben instalar herramientas como *Cabal* y *Stack* . Si está ejecutando un sistema operativo similar a Unix, la instalación más sencilla es instalar *Stack* utilizando:

```
curl -sSL https://get.haskellstack.org/ | sh
```

Esto instala GHC aislado del resto de su sistema, por lo que es fácil de eliminar. Sin embargo, todos los comandos deben ir precedidos de la `stack` . Otro enfoque simple es instalar una plataforma Haskell . La plataforma existe en dos sabores:

1. La distribución **mínima** contiene solo *GHC* (para compilar) y *Cabal* / *Stack* (para instalar y construir paquetes)
2. La distribución **completa** además contiene herramientas para el desarrollo de proyectos, perfiles y análisis de cobertura. También se incluye un conjunto adicional de paquetes ampliamente utilizados.

Estas plataformas se pueden instalar descargando un instalador y siguiendo las instrucciones o usando el administrador de paquetes de su distribución (tenga en cuenta que no se garantiza que esta versión esté actualizada):

- Ubuntu, Debian, Mint:

  ```
  sudo apt-get install haskell-platform
  ```

- Fedora:

  ```
  sudo dnf install haskell-platform
  ```

- Sombrero rojo:

  ```
  sudo yum install haskell-platform
  ```

- Arch Linux:

  ```
  sudo pacman -S ghc cabal-install haskell-haddock-api \
                 haskell-haddock-library happy alex
  ```

- Gentoo:

  ```
  sudo layman -a haskell
  sudo emerge haskell-platform
  ```

- OSX con Homebrew:

```
brew cask install haskell-platform
```

- OSX con MacPorts:

```
sudo port install haskell-platform
```

Una vez instalado, debería ser posible iniciar *GHCi* invocando el comando `ghci` en cualquier parte del terminal. Si la instalación salió bien, la consola debería verse como

```
me@notebook:~$ ghci
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Prelude>
```

posiblemente con más información sobre qué bibliotecas se han cargado antes del `Prelude>` . Ahora, la consola se ha convertido en una REPL de Haskell y puede ejecutar el código de Haskell como en la REPL en línea. Para salir de este entorno interactivo, se puede escribir `:q` o `:quit` . Para obtener más información sobre qué comandos están disponibles en *GHCi* , escriba `:?` como se indica en la pantalla de inicio.

Debido a que escribir las mismas cosas una y otra vez en una sola línea no siempre es tan práctico, podría ser una buena idea escribir el código de Haskell en archivos. Estos archivos normalmente tienen `.hs` para una extensión y se pueden cargar en el REPL usando `:l` o `:load` .

Como se mencionó anteriormente, *GHCi* es una parte del *GHC* , que en realidad es un compilador. Este compilador se puede usar para transformar un archivo `.hs` con código Haskell en un programa en ejecución. Debido a que un archivo `.hs` puede contener muchas funciones, se debe definir una función `main` en el archivo. Este será el punto de partida del programa. El archivo `test.hs` se puede compilar con el comando

```
ghc test.hs
```

esto creará archivos de objetos y un ejecutable si no hubo errores y la función `main` se definió correctamente.

Herramientas mas avanzadas

1. Ya se mencionó anteriormente como administrador de paquetes, pero la *pila* puede ser una herramienta útil para el desarrollo de Haskell de maneras completamente diferentes. Una vez instalado, es capaz de

    - Instalación (múltiples versiones de) *GHC*
    - Creación de proyectos y andamios.
    - gestión de dependencias
    - proyectos de construcción y pruebas

- evaluación comparativa

2. IHaskell es un núcleo de haskell para IPython y permite combinar código (ejecutable) con markdown y notación matemática.

Primes

Algunas variantes *más sobresalientes* :

Por debajo de 100

```
import Data.List ( (\\) )

ps100 = ((([2..100] \\ [4,6..100]) \\ [6,9..100]) \\ [10,15..100]) \\ [14,21..100]

   -- = (((2:[3,5..100]) \\ [9,15..100]) \\ [25,35..100]) \\ [49,63..100]

   -- = (2:[3,5..100]) \\ ([9,15..100] ++ [25,35..100] ++ [49,63..100])
```

Ilimitado

Tamiz de Eratóstenes, usando el paquete de datos-ordlist :

```
import qualified Data.List.Ordered

ps   = 2 : _Y ((3:) . minus [5,7..] . unionAll . map (\p -> [p*p, p*p+2*p..]))

_Y g = g (_Y g)   -- = g (g (_Y g)) = g (g (g (g (...)))) = g . g . g . g . ...
```

Tradicional

(un tamiz de división de prueba sub-óptimo)

```
ps = sieve [2..]
     where
     sieve (x:xs) = [x] ++ sieve [y | y <- xs, rem y x > 0]

-- = map head ( iterate (\(x:xs) -> filter ((> 0).(`rem` x)) xs) [2..] )
```

División de prueba óptima

```
ps = 2 : [n | n <- [3..], all ((> 0).rem n) $ takeWhile ((<= n).(^2)) ps]

-- = 2 : [n | n <- [3..], foldr (\p r-> p*p > n || (rem n p > 0 && r)) True ps]
```

Transicional

De la división de prueba al tamiz de Eratóstenes:

```
[n | n <- [2..], []==[i | i <- [2..n-1], j <- [0,i..n], j==n]]
```

El código más corto

```
nubBy (((>1).).gcd) [2..]          -- i.e., nubBy (\a b -> gcd a b > 1) [2..]
```

`nubBy` también es de `Data.List` , como `(\\)` .

Declarar valores

Podemos declarar una serie de expresiones en el REPL así:

```
Prelude> let x = 5
Prelude> let y = 2 * 5 + x
Prelude> let result = y * 10
Prelude> x
5
Prelude> y
15
Prelude> result
150
```

Para declarar los mismos valores en un archivo escribimos lo siguiente:

```
-- demo.hs

module Demo where
-- We declare the name of our module so
-- it can be imported by name in a project.

x = 5

y = 2 * 5 + x

result = y * 10
```

Los nombres de los módulos están en mayúsculas, a diferencia de los nombres de variables.

Lea Empezando con Haskell Language en línea:
https://riptutorial.com/es/haskell/topic/251/empezando-con-haskell-language

Capítulo 2: Agujeros mecanografiados

Observaciones

Una de las fortalezas de Haskell es la capacidad de aprovechar el sistema de tipos para modelar partes de su dominio de problemas en el sistema de tipos. Al hacerlo, a menudo se encuentran tipos muy complejos. Cuando se escriben programas con estos tipos (es decir, con valores que tienen estos tipos) ocasionalmente se vuelve casi inmutable para "hacer malabares" con todos los tipos. A partir de GHC 7.8, hay una nueva característica sintáctica llamada agujeros mecanografiados. Los orificios escritos no cambian la semántica del lenguaje central; están destinados puramente como una ayuda para la escritura de programas.

Para obtener una explicación detallada de los agujeros escritos, así como una discusión sobre el diseño de los agujeros escritos, consulte la wiki de Haskell .

Sección de la guía del usuario de GHC sobre agujeros mecanografiados .

Examples

Sintaxis de los agujeros mecanografiados.

Un agujero escrito es un guión bajo (_) o un identificador de Haskell válido que no está en el alcance, en un contexto de expresión. Antes de la existencia de orificios escritos, ambas cosas podrían provocar un error, por lo que la nueva sintaxis no interfiere con ninguna sintaxis antigua.

Controlando el comportamiento de los agujeros mecanografiados.

El comportamiento predeterminado de los agujeros escritos es producir un error en tiempo de compilación cuando se encuentra un agujero escrito. Sin embargo, hay varias banderas para afinar su comportamiento. Estas banderas se resumen de la siguiente manera (trac de GHC):

> Por defecto, GHC ha habilitado los orificios y ha producido un error de compilación cuando encuentra un orificio escrito.
>
> Cuando `-fdefer-type-errors` **o** `-fdefer-typed-holes` está habilitado, los errores de los hoyos se convierten en advertencias y resultan en errores de tiempo de ejecución cuando se evalúan.
>
> El indicador de advertencia `-fwarn-typed-holes` está `-fwarn-typed-holes` forma predeterminada. Sin los `-fdefer-type-errors` `-fdefer-typed-holes` o los `-fdefer-typed-holes` esta bandera no es `-fdefer-typed-holes` , ya que los agujeros escritos son un error en estas condiciones. Si cualquiera de las banderas de aplazamiento está habilitada (convirtiendo los errores de orificio escritos en advertencias), la bandera de

`-fno-warn-typed-holes` desactiva las advertencias. Esto significa que la compilación se realiza de forma silenciosa y la evaluación de un agujero producirá un error de tiempo de ejecución.

Semántica de agujeros mecanografiados.

Se puede decir simplemente que el valor de un agujero de tipo `undefined` está `undefined` , aunque un agujero escrito genera un error en tiempo de compilación, por lo que no es estrictamente necesario asignarle un valor. Sin embargo, un agujero escrito (cuando están habilitados) produce un error de tiempo de compilación (o advertencia con errores de tipo diferido) que indica el nombre del agujero escrito, su tipo *más general* inferido y los tipos de enlaces locales. Por ejemplo:

```
Prelude> \x -> _var + length (drop 1 x)

<interactive>:19:7: Warning:
    Found hole `_var' with type: Int
    Relevant bindings include
      x :: [a] (bound at <interactive>:19:2)
      it :: [a] -> Int (bound at <interactive>:19:1)
    In the first argument of `(+)', namely `_var'
    In the expression: _var + length (drop 1 x)
    In the expression: \ x -> _var + length (drop 1 x)
```

Tenga en cuenta que en el caso de los orificios escritos en las expresiones ingresadas en la respuesta de GHCi (como anteriormente), también se informó el tipo de expresión ingresada, ya `it` (aquí de tipo `[a] -> Int`).

Usando agujeros escritos para definir una instancia de clase

Los orificios escritos pueden facilitar la definición de funciones, a través de un proceso interactivo.

Supongamos que desea definir una instancia de clase `Foo Bar` (para su tipo de `Bar` personalizado, para usarla con alguna función de biblioteca polimórfica que requiere una instancia de `Foo`). Ahora, tradicionalmente, buscaría la documentación de `Foo` , descubriría qué métodos necesita definir, escrutar sus tipos, etc. - pero con los agujeros tipificados, puede omitir eso.

Primero solo define una instancia ficticia:

```
instance Foo Bar where
```

El compilador ahora se quejará

```
Bar.hs:13:10: Warning:
No explicit implementation for
  ‘foom’ and ‘quun’
In the instance declaration for ‘Foo Bar’
```

Ok, entonces necesitamos definir `foom` para `Bar` . ¿Pero qué *se* supone que *es* eso? Una vez más, somos demasiado perezosos para mirar en la documentación, y solo pregunte al compilador:

```
instance Foo Bar where
  foom = _
```

Aquí hemos utilizado un agujero escrito como una simple "consulta de documentación". Las salidas del compilador

```
Bar.hs:14:10:
    Found hole ‘_’ with type: Bar -> Gronk Bar
    Relevant bindings include
      foom :: Bar -> Gronk Bar (bound at Foo.hs:4:28)
    In the expression: _
    In an equation for ‘foom’: foom = _
    In the instance declaration for ‘Foo Bar’
```

Observe cómo el compilador ya ha llenado la variable de tipo de clase con la `Bar` tipo concreto para la que queremos crear una instancia. Esto puede hacer que la firma sea mucho más fácil de entender que la polimórfica que se encuentra en la documentación de la clase, especialmente si se trata de un método más complicado, por ejemplo, una clase de tipo de múltiples parámetros.

¿Pero qué diablos es `Gronk` ? En este punto, probablemente sea una buena idea preguntarle a Hayoo . Sin embargo, aún podemos evitarlo: como una suposición ciega, asumimos que esto no es solo un constructor de tipo sino también el constructor de un solo valor, es decir, se puede usar como una función que de alguna manera producirá `Gronk a` valor `Gronk a` . Así que intentamos

```
instance Foo Bar where
  foom bar = _ Gronk
```

Si tenemos suerte, `Gronk` es en realidad un valor, y el compilador ahora dirá

```
    Found hole ‘_’
      with type: (Int -> [(Int, b0)] -> Gronk b0) -> Gronk Bar
    Where: ‘b0’ is an ambiguous type variable
```

Ok, eso es feo. Al principio solo note que `Gronk` tiene dos argumentos, así que podemos refinar nuestro intento:

```
instance Foo Bar where
  foom bar = Gronk _ _
```

Y esto ahora es bastante claro:

```
    Found hole ‘_’ with type: [(Int, Bar)]
    Relevant bindings include
      bar :: Bar (bound at Bar.hs:14:29)
      foom :: Bar -> Gronk Bar (bound at Foo.hs:15:24)
    In the second argument of ‘Gronk’, namely ‘_’
    In the expression: Gronk _ _
    In an equation for ‘foom’: foom bar = Gronk _ _
```

Ahora puede seguir avanzando, por ejemplo, deconstruyendo el valor de la `bar` (los componentes se mostrarán, con tipos, en la sección `Relevant bindings`). A menudo, en algún punto es

completamente obvio cuál será la definición correcta, porque ves todos los argumentos disponibles y los tipos encajan como un rompecabezas. O alternativamente, puedes ver que la definición es *imposible* y por qué.

Todo esto funciona mejor en un editor con compilación interactiva, por ejemplo, Emacs con modo haskell. A continuación, puede usar orificios escritos de forma muy parecida a las consultas de valor de mouse-over en un IDE para un lenguaje imperativo dinámico interpretado, pero sin todas las limitaciones.

Lea Agujeros mecanografiados en línea: https://riptutorial.com/es/haskell/topic/4913/agujeros-mecanografiados

Capítulo 3: Analizando HTML con lentes etiquetadas y lentes

Examples

Extrae el contenido del texto de un div con un id particular

Taggy-lens nos permite usar lentes para analizar e inspeccionar documentos HTML.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.0 --install-ghc runghc --package text --package lens --package
taggy-lens

{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text.Lazy as TL
import qualified Data.Text.IO as T
import Text.Taggy.Lens
import Control.Lens

someHtml :: TL.Text
someHtml =
    "\
    \<!doctype html><html><body>\
    \<div>first div</div>\
    \<div id=\"thediv\">second div</div>\
    \<div id=\"not-thediv\">third div</div>"

main :: IO ()
main = do
    T.putStrLn
        (someHtml ^. html . allAttributed (ix "id" . only "thediv") . contents)
```

Filtrar elementos del árbol.

Encuentre div con `id="article"` y elimine todas las etiquetas de script internas.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.1 --install-ghc runghc --package text --package lens --package
taggy-lens --package string-class --package classy-prelude
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}

import ClassyPrelude
import Control.Lens hiding (children, element)
import Data.String.Class (toText, fromText, toString)
import Data.Text (Text)
import Text.Taggy.Lens
import qualified Text.Taggy.Lens as Taggy
import qualified Text.Taggy.Renderer as Renderer

somehtmlSmall :: Text
```

```
somehtmlSmall =
    "<!doctype html><html><body>\
    \<div id=\"article\"><div>first</div><div>second</div><script>this should be
removed</script><div>third</div></div>\
    \</body></html>"

renderWithoutScriptTag :: Text
renderWithoutScriptTag =
    let mArticle :: Maybe Taggy.Element
        mArticle =
            (fromText somehtmlSmall) ^? html .
            allAttributed (ix "id" . only "article")
        mArticleFiltered =
            fmap
                (transform
                     (children %~
                      filter (\n -> n ^? element . name /= Just "script")))
                mArticle
    in maybe "" (toText . Renderer.render) mArticleFiltered

main :: IO ()
main = print renderWithoutScriptTag
-- outputs:
-- "<div id=\"article\"><div>first</div><div>second</div><div>third</div></div>"
```

Contribución basada en la respuesta SO de @ duplode

Lea Analizando HTML con lentes etiquetadas y lentes en línea:
https://riptutorial.com/es/haskell/topic/6962/analizando-html-con-lentes-etiquetadas-y-lentes

Capítulo 4: Apilar

Examples

Instalación de la pila

Mac OS X

Usando Homebrew :

```
brew install haskell-stack
```

Creando un proyecto simple

Para crear un proyecto llamado **helloworld** run:

```
stack new helloworld simple
```

Esto creará un directorio llamado `helloworld` con los archivos necesarios para un proyecto de pila.

Estructura

Estructura de archivos

Un proyecto simple tiene los siguientes archivos incluidos en él:

```
➜ helloworld ls
LICENSE        Setup.hs        helloworld.cabal src            stack.yaml
```

En la carpeta `src` hay un archivo llamado `Main.hs` Este es el "punto de partida" del proyecto `helloworld` . Por defecto, `Main.hs` contiene un simple "¡Hola mundo!" programa.

Main.hs

```
module Main where

main :: IO ()
main = do
  putStrLn "hello world"
```

Ejecutando el programa

Asegúrate de estar en el directorio `helloworld` y ejecuta:

```
stack build # Compile the program
stack exec helloworld # Run the program
# prints "hello world"
```

Paquetes de apilamiento y cambio de la versión LTS (resolución)

Stackage es un repositorio de paquetes de Haskell. Podemos agregar estos paquetes a un proyecto de pila.

Añadiendo lentes a un proyecto.

En un proyecto de pila, hay un archivo llamado `stack.yaml` . En `stack.yaml` hay un segmento que se parece a:

```
resolver: lts-6.8
```

Stackage mantiene una lista de paquetes para cada revisión de `lts` . En nuestro caso, queremos la lista de paquetes para `lts-6.8` Para encontrar estos paquetes, visite:

```
https://www.stackage.org/lts-6.8 # if a different version is used, change 6.8 to the correct
resolver number.
```

Mirando a través de los paquetes, hay un Lens-4.13 .

Ahora podemos agregar el paquete de idioma modificando la sección de `helloworld.cabal` :

```
  build-depends: base >= 4.7 && < 5
```

a:

```
  build-depends: base >= 4.7 && 5,
                 lens == 4.13
```

Obviamente, si queremos cambiar un LTS más nuevo (después de su lanzamiento), simplemente cambiamos el número de resolución, por ejemplo:

```
resolver: lts-6.9
```

Con la siguiente `stack build` pila utilizará la versión LTS 6.9 y, por lo tanto, descargará algunas nuevas dependencias.

Construir y ejecutar un proyecto de pila

En este ejemplo, nuestro nombre de proyecto es "helloworld", que se creó con `stack new helloworld simple`

Primero tenemos que construir el proyecto con la `stack build` y luego podemos ejecutarlo con

```
stack exec helloworld-exe
```

Instalación de pila

Ejecutando el comando

```
stack install
```

Stack copiará un archivo ejecutable a la carpeta.

```
/Users/<yourusername>/.local/bin/
```

Perfilando con Stack

Configurar perfiles para un proyecto a través de la `stack` . Primero construye el proyecto con la bandera `--profile` :

```
stack build --profile
```

Las banderas de GHC no se requieren en el archivo cabal para que esto funcione (como `-prof`). `stack` activará automáticamente el perfil tanto para la biblioteca como para los ejecutables en el proyecto. La próxima vez que se ejecute un ejecutable en el proyecto, se pueden usar los indicadores `+RTS` habituales:

```
stack exec -- my-bin +RTS -p
```

Visualización de dependencias

Para averiguar de qué paquetes depende su proyecto directamente, simplemente puede usar este comando:

```
stack list-dependencies
```

De esta manera, puede averiguar qué versión de sus dependencias realmente se eliminó por pila.

Los proyectos de Haskell a menudo se encuentran tirando de muchas bibliotecas de manera indirecta, y algunas veces estas dependencias externas causan problemas que necesita rastrear. Si se encuentra con una dependencia externa deshonesta que le gustaría identificar, puede revisar todo el gráfico de dependencias e identificar cuál de sus dependencias está finalmente obteniendo el paquete no deseado:

```
stack dot --external | grep template-haskell
```

`stack dot` imprime un gráfico de dependencia en forma de texto que se puede buscar. También se puede ver:

```
stack dot --external | dot -Tpng -o my-project.png
```

También puede establecer la profundidad del gráfico de dependencia si desea:

```
stack dot --external --depth 3 | dot -Tpng -o my-project.png
```

Lea Apilar en línea: https://riptutorial.com/es/haskell/topic/2970/apilar

Capítulo 5: Aritmética

Introducción

En Haskell, todas las expresiones (que incluyen las constantes numéricas y las funciones que operan en ellas) tienen un tipo decidible. En el momento de la compilación, el comprobador de tipos infiere el tipo de una expresión de los tipos de funciones elementales que la componen. Dado que los datos son inmutables de manera predeterminada, no hay operaciones de "conversión de tipo", pero hay funciones que copian los datos y generalizan o especializan los tipos dentro de la razón.

Observaciones

La jerarquía de clases de tipo numérico.

`Num` encuentra en la raíz de la jerarquía de clases de tipo numérica. A continuación se muestran sus operaciones características y algunas instancias comunes (las cargadas de forma predeterminada con Prelude más las de `Data.Complex`):

```
λ> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
      -- Defined in ‘GHC.Num’
instance RealFloat a => Num (Complex a) -- Defined in ‘Data.Complex’
instance Num Word -- Defined in ‘GHC.Num’
instance Num Integer -- Defined in ‘GHC.Num’
instance Num Int -- Defined in ‘GHC.Num’
instance Num Float -- Defined in ‘GHC.Float’
instance Num Double -- Defined in ‘GHC.Float’
```

Ya hemos visto la clase `Fractional` , que requiere `Num` e introduce las nociones de "división" `(/)` y recíproco de un número:

```
λ> :i Fractional
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
      -- Defined in ‘GHC.Real’
instance RealFloat a => Fractional (Complex a) -- Defined in ‘Data.Complex’
instance Fractional Float -- Defined in ‘GHC.Float’
```

```
instance Fractional Double -- Defined in ‘GHC.Float’
```

La clase `Real` modela ... los números reales. Requiere `Num` y `Ord` , por lo tanto, modela un campo numérico ordenado. Como contraejemplo, los números complejos *no* son un campo ordenado (es decir, no poseen una relación de orden natural):

```
λ> :i Real
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
  {-# MINIMAL toRational #-}
      -- Defined in ‘GHC.Real’
instance Real Word -- Defined in ‘GHC.Real’
instance Real Integer -- Defined in ‘GHC.Real’
instance Real Int -- Defined in ‘GHC.Real’
instance Real Float -- Defined in ‘GHC.Float’
instance Real Double -- Defined in ‘GHC.Float’
```

`RealFrac` representa números que pueden ser redondeados

```
λ> :i RealFrac
class (Real a, Fractional a) => RealFrac a where
  properFraction :: Integral b => a -> (b, a)
  truncate :: Integral b => a -> b
  round :: Integral b => a -> b
  ceiling :: Integral b => a -> b
  floor :: Integral b => a -> b
  {-# MINIMAL properFraction #-}
      -- Defined in ‘GHC.Real’
instance RealFrac Float -- Defined in ‘GHC.Float’
instance RealFrac Double -- Defined in ‘GHC.Float’
```

`Floating` (lo que implica `Fractional`) representa constantes y operaciones que pueden no tener una expansión decimal finita.

```
λ> :i Floating
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  sqrt :: a -> a
  (**) :: a -> a -> a
  logBase :: a -> a -> a
  sin :: a -> a
  cos :: a -> a
  tan :: a -> a
  asin :: a -> a
  acos :: a -> a
  atan :: a -> a
  sinh :: a -> a
  cosh :: a -> a
  tanh :: a -> a
  asinh :: a -> a
  acosh :: a -> a
  atanh :: a -> a
  GHC.Float.log1p :: a -> a
  GHC.Float.expm1 :: a -> a
```

```
  GHC.Float.log1pexp :: a -> a
  GHC.Float.log1mexp :: a -> a
  {-# MINIMAL pi, exp, log, sin, cos, asin, acos, atan, sinh, cosh,
              asinh, acosh, atanh #-}
      -- Defined in ‘GHC.Float’
instance RealFloat a => Floating (Complex a) -- Defined in ‘Data.Complex’
instance Floating Float -- Defined in ‘GHC.Float’
instance Floating Double -- Defined in ‘GHC.Float’
```

Precaución: mientras que expresiones como `sqrt . negate :: Floating a => a -> a` son perfectamente válidos, pueden devolver `NaN` ("no-a-number"), que puede no ser un comportamiento intencionado. En tales casos, podríamos querer trabajar sobre el campo Complejo (se muestra más adelante).

Examples

Ejemplos basicos

```
λ> :t 1
1 :: Num t => t

λ> :t pi
pi :: Floating a => a
```

En los ejemplos anteriores, el comprobador de tipos infiere una *clase de* tipo en lugar de un tipo concreto para las dos constantes. En Haskell, la clase `Num` es la más numérica general (ya que abarca números enteros y reales), pero `pi` debe pertenecer a una clase más especializada, ya que tiene una parte fraccionaria distinta de cero.

```
list0 :: [Integer]
list0 = [1, 2, 3]

list1 :: [Double]
list1 = [1, 2, pi]
```

Los tipos de hormigón anteriores fueron inferidos por GHC. Tipos más generales como `list0 :: Num a => [a]` habrían funcionado, pero también hubieran sido más difíciles de conservar (por ejemplo, si uno considera un `Double` en una lista de `Num` s), debido a las advertencias que se muestran arriba.

`No se pudo deducir (Fraccional Int) ...`

El mensaje de error en el título es un error común de principiante. Veamos cómo surge y cómo solucionarlo.

Supongamos que necesitamos calcular el valor promedio de una lista de números; La siguiente declaración parecería hacerlo, pero no compilaría:

```
averageOfList ll = sum ll / length ll
```

El problema es con la función de división `(/)` : su firma es `(/) :: Fractional a => a -> a -> a` , pero en el caso sobre el denominador (dado por `length :: Foldable t => ta -> Int`) es de tipo `Int` (y `Int` no pertenece a la clase `Fractional`) de ahí el mensaje de error.

Podemos corregir el mensaje de error con `fromIntegral :: (Num b, Integral a) => a -> b` . Se puede ver que esta función acepta valores de cualquier tipo `Integral` y devuelve los correspondientes en la clase `Num` :

```
averageOfList' :: (Foldable t, Fractional a) => t a -> a
averageOfList' ll = sum ll / fromIntegral (length ll)
```

Ejemplos de funciones

¿Cuál es el tipo de `(+)` ?

```
λ> :t (+)
(+) :: Num a => a -> a -> a
```

¿Cuál es el tipo de `sqrt` ?

```
λ> :t sqrt
sqrt :: Floating a => a -> a
```

¿Cuál es el tipo de `sqrt . fromIntegral` ?

```
sqrt . fromIntegral :: (Integral a, Floating c) => a -> c
```

Lea Aritmética en línea: https://riptutorial.com/es/haskell/topic/8616/aritmetica

Capítulo 6: Atravesable

Introducción

La clase `Traversable` generaliza la función anteriormente conocida como `mapM :: Monad m => (a -> mb) -> [a] -> m [b]` para trabajar con efectos `Applicative` sobre estructuras distintas a las listas.

Examples

Funcionalizador de funciones y plegable para una estructura transversal

```
import Data.Traversable as Traversable

data MyType a =  -- ...
instance Traversable MyType where
    traverse = -- ...
```

Cada estructura de `Traversable` se puede convertir en un `Foldable Functor` usando las funciones `fmapDefault` y `foldMapDefault` que se encuentran en `Data.Traversable` .

```
instance Functor MyType where
    fmap = Traversable.fmapDefault

instance Foldable MyType where
    foldMap = Traversable.foldMapDefault
```

`fmapDefault` se define ejecutando el `traverse` en el functor aplicativo de `Identity` .

```
newtype Identity a = Identity { runIdentity :: a }

instance Applicative Identity where
    pure = Identity
    Identity f <*> Identity x = Identity (f x)

fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault f = runIdentity . traverse (Identity . f)
```

`foldMapDefault` se define utilizando el functor aplicativo `Const` , que ignora su parámetro mientras acumula un valor monoidal.

```
newtype Const c a = Const { getConst :: c }

instance Monoid m => Applicative (Const m) where
    pure _ = Const mempty
    Const x <*> Const y = Const (x `mappend` y)

foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
foldMapDefault f = getConst . traverse (Const . f)
```

Una instancia de Traversable para un árbol binario

Las implementaciones de `traverse` generalmente se ven como una implementación de `fmap` levantada en un contexto `Applicative` .

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

instance Traversable Tree where
    traverse f Leaf = pure Leaf
    traverse f (Node l x r) = Node <$> traverse f l <*> f x <*> traverse f r
```

Esta implementación realiza un recorrido en orden del árbol.

```
ghci> let myTree = Node (Node Leaf 'a' Leaf) 'b' (Node Leaf 'c' Leaf)

--    +--'b'--+
--    |       |
-- +-'a'-+ +-'c'-+
-- |     | |     |
-- *     * *     *

ghci> traverse print myTree
'a'
'b'
'c'
```

La extensión `DeriveTraversable` permite a GHC generar instancias `Traversable` basadas en la estructura del tipo. Podemos variar el orden del recorrido escrito por la máquina ajustando el diseño del constructor `Node` .

```
data Inorder a = ILeaf
               | INode (Inorder a) a (Inorder a)  -- as before
               deriving (Functor, Foldable, Traversable)  -- also using DeriveFunctor and
DeriveFoldable

data Preorder a = PrLeaf
                | PrNode a (Preorder a) (Preorder a)
                deriving (Functor, Foldable, Traversable)

data Postorder a = PoLeaf
                 | PoNode (Postorder a) (Postorder a) a
                 deriving (Functor, Foldable, Traversable)

-- injections from the earlier Tree type
inorder :: Tree a -> Inorder a
inorder Leaf = ILeaf
inorder (Node l x r) = INode (inorder l) x (inorder r)

preorder :: Tree a -> Preorder a
preorder Leaf = PrLeaf
preorder (Node l x r) = PrNode x (preorder l) (preorder r)

postorder :: Tree a -> Postorder a
postorder Leaf = PoLeaf
postorder (Node l x r) = PoNode (postorder l) (postorder r) x
```

```
ghci> traverse print (inorder myTree)
'a'
'b'
'c'
ghci> traverse print (preorder myTree)
'b'
'a'
'c'
ghci> traverse print (postorder myTree)
'a'
'c'
'b'
```

Atravesando una estructura en reversa

Un recorrido se puede ejecutar en la dirección opuesta con la ayuda del funtor aplicativo `Backwards` , que invierte un aplicativo existente para que los efectos compuestos se realicen en orden inverso.

```
newtype Backwards f a = Backwards { forwards :: f a }

instance Applicative f => Applicative (Backwards f) where
    pure = Backwards . pure
    Backwards ff <*> Backwards fx = Backwards ((\x f -> f x) <$> fx <*> ff)
```

`Backwards` se puede poner en uso en una " `traverse` invertida". Cuando el aplicativo subyacente de una llamada `traverse` se invierte con `Backwards` , el efecto resultante sucede en orden inverso.

```
newtype Reverse t a = Reverse { getReverse :: t a }

instance Traversable t => Traversable (Reverse t) where
    traverse f = fmap Reverse . forwards . traverse (Backwards . f) . getReverse

ghci> traverse print (Reverse "abc")
'c'
'b'
'a'
```

El `Reverse` newtype se encuentra en Data.Functor.Reverse.

Definición de Traversable

```
class (Functor t, Foldable t) => Traversable t where
    {-# MINIMAL traverse | sequenceA #-}

    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    traverse f = sequenceA . fmap f

    sequenceA :: Applicative f => t (f a) -> f (t a)
    sequenceA = traverse id

    mapM :: Monad m => (a -> m b) -> t a -> m (t b)
    mapM = traverse
```

```
    sequence :: Monad m => t (m a) -> m (t a)
    sequence = sequenceA
```

`Traversable` estructuras `t` son contenedores finitistas de elementos `a` que se pueden accionar con una operación effectful "visitante". La función de visitante `f :: a -> fb` realiza un efecto secundario en cada elemento de la estructura y la `traverse` compone los efectos secundarios utilizando el `Applicative` . Otra forma de verlo es que la `sequenceA` dice que las estructuras `Traversable` se conmutan con el `Applicative` s.

Transformación de una estructura transitable con la ayuda de un parámetro de acumulación

Las dos funciones `mapAccum` combinan las operaciones de plegado y mapeo.

```
--                                                    A Traversable structure
--                                                                 |
--                                                  A seed value   |
--                                                          |      |
--                                                         |-|   |---|
mapAccumL, mapAccumR :: Traversable t => (a -> b -> (a, c)) -> a -> t b -> (a, t c)
--                                       |------------------|              |--------|
--                                                  |                           |
--                        A folding function which produces a new mapped        |
--                          element 'c' and a new accumulator value 'a'         |
--                                                                              |
--                                                             Final accumulator value
--                                                              and mapped structure
```

Estas funciones generalizan `fmap` en el `fmap` de que permiten que los valores asignados dependan de lo que sucedió anteriormente en el pliegue. Generalizan `foldl` / `foldr` en el `foldr` de que mapean la estructura en su lugar y la reducen a un valor.

Por ejemplo, `tails` pueden ser implementados utilizando `mapAccumR` y su hermana `inits` pueden implementarse utilizando `mapAccumL` .

```
tails, inits :: [a] -> [[a]]
tails = uncurry (:) . mapAccumR (\xs x -> (x:xs, xs)) []
inits = uncurry snoc . mapAccumL (\xs x -> (x `snoc` xs, xs)) []
    where snoc x xs = xs ++ [x]

ghci> tails "abc"
["abc", "bc", "c", ""]
ghci> inits "abc"
["", "a", "ab", "abc"]
```

`mapAccumL` se implementa al atravesar el funtor aplicativo del `State` .

```
{-# LANGUAGE DeriveFunctor #-}

newtype State s a = State { runState :: s -> (s, a) } deriving Functor
instance Applicative (State s) where
    pure x = State $ \s -> (s, x)
```

```
    State ff <*> State fx = State $ \s -> let (t, f) = ff s
                                              (u, x) = fx t
                                          in (u, f x)

mapAccumL f z t = runState (traverse (State . flip f) t) z
```

`mapAccumR` funciona ejecutando `mapAccumL` en sentido inverso .

```
mapAccumR f z = fmap getReverse . mapAccumL f z . Reverse
```

Estructuras transitables como formas con contenidos.

Si un tipo `t` es `Traversable` , los valores de `ta` se pueden dividir en dos partes: su "forma" y su "contenido":

```
data Traversed t a = Traversed { shape :: t (), contents :: [a] }
```

donde los "contenidos" son los mismos que los que "visitaría" utilizando una instancia `Foldable` .

Ir en una dirección, de `ta` a `Traversed ta` no requiere nada más que `Functor` y `Foldable`

```
break :: (Functor t, Foldable t) => t a -> Traversed t a
break ta = Traversed (fmap (const ()) ta) (toList ta)
```

Pero volviendo atrás usa la función `traverse` crucialmente.

```
import Control.Monad.State

-- invariant: state is non-empty
pop :: State [a] a
pop = state $ \(a:as) -> (a, as)

recombine :: Traversable t => Traversed t a -> t a
recombine (Traversed s c) = evalState (traverse (const pop) s) c
```

Las leyes de `Traversable` requieren esa `break . recombine` y `recombine . break` son ambas identidades En particular, esto significa que hay exactamente los elementos numéricos correctos en el `contents` para rellenar la `shape` completamente sin dejar sobras.

`Traversed t` es `Traversable` sí mismo. La implementación de `traverse` funciona visitando los elementos utilizando la instancia de `Traversable` de la lista y luego volviendo a unir la forma inerte al resultado.

```
instance Traversable (Traversed t) where
    traverse f (Traversed s c) = fmap (Traversed s) (traverse f c)
```

Transponer una lista de listas

Teniendo en cuenta que `zip` transpone una tupla de listas en una lista de tuplas,

```
ghci> uncurry zip ([1,2],[3,4])
[(1,3), (2,4)]
```

y la similitud entre los tipos de `transpose` y `sequenceA` ,

```
-- transpose exchanges the inner list with the outer list
--           +---+--><--+-+
--           |   |     | |
transpose :: [[a]] -> [[a]]
--            | |     |   |
--            +-+--><--+---+

-- sequenceA exchanges the inner Applicative with the outer Traversable
--                                            +------>------+
--                                            |             |
sequenceA :: (Traversable t, Applicative f) => t (f a) -> f (t a)
--                                              |       |
--                                              +--->---+
```

la idea es usar la estructura de `Traversable` y `Applicative []` para desplegar la `sequenceA` como una especie de *`zip` n-ary* , uniendo todas las listas internas de manera puntual.

`[]` 'Elección prioridad' 's por defecto `Applicative` instancia no es apropiado para nuestro uso - necesitamos un 'enérgico' `Applicative` . Para esto utilizamos el `ZipList` ZipList, que se encuentra en `Control.Applicative` .

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
    pure x = ZipList (repeat x)
    ZipList fs <*> ZipList xs = ZipList (zipWith ($) fs xs)
```

Ahora tenemos `transpose` de forma gratuita, por la que atraviesa en el `ZipList Applicative` .

```
transpose :: [[a]] -> [[a]]
transpose = getZipList . traverse ZipList

ghci> let myMatrix = [[1,2,3],[4,5,6],[7,8,9]]
ghci> transpose myMatrix
[[1,4,7],[2,5,8],[3,6,9]]
```

Lea Atravesable en línea: https://riptutorial.com/es/haskell/topic/754/atravesable

Capítulo 7: Attoparsec

Introducción

Attoparsec es una biblioteca combinadora de análisis que está "dirigida especialmente a tratar de manera eficiente los protocolos de red y los complicados formatos de archivo de texto / binario".

Attoparsec ofrece no solo velocidad y eficiencia, sino también retroceso e ingreso incremental.

Su API es muy similar a la de otra biblioteca combinadora de analizadores, Parsec.

Hay submódulos para la compatibilidad con `ByteString` , `Text` y `Char8` . Se recomienda el uso de la extensión de lenguaje `OverloadedStrings` .

Parámetros

Tipo	Detalle
`Parser ia`	El tipo de núcleo para representar un analizador. `i` es el tipo de cadena, por ejemplo, `ByteString` .
`IResult ir`	El resultado de un análisis, con `Fail i [String] String` , `Partial (i -> IResult ir)` y `Done ir` como constructores.

Examples

Combinadores

La entrada de análisis se logra mejor a través de funciones de analizador más grandes que se componen de funciones más pequeñas y de un solo propósito.

Digamos que deseamos analizar el siguiente texto que representa las horas de trabajo:

> Lunes: 0800 1600.

Podríamos dividirlos en dos "tokens": el nombre del día, "Monday", y una parte de tiempo "0800" a "1600".

Para analizar el nombre de un día, podríamos escribir lo siguiente:

```
data Day = Day String

day :: Parser Day
day = do
  name <- takeWhile1 (/= ':')
  skipMany1 (char ':')
```

```
  skipSpace
  return $ Day name
```

Para analizar la parte del tiempo podríamos escribir:

```
data TimePortion = TimePortion String String

time = do
    start <- takeWhile1 isDigit
    skipSpace
    end <- takeWhile1 isDigit
    return $ TimePortion start end
```

Ahora que tenemos dos analizadores para nuestras partes individuales del texto, podemos combinarlos en un analizador "más grande" para leer las horas de trabajo de un día entero:

```
data WorkPeriod = WorkPeriod Day TimePortion

work = do
    d <- day
    t <- time
    return $ WorkPeriod d t
```

y luego ejecute el analizador

```
parseOnly work "Monday: 0800 1600"
```

Mapa de bits - Análisis de datos binarios

Attoparsec hace que el análisis de datos binarios sea trivial. Asumiendo estas definiciones:

```
import           Data.Attoparsec.ByteString (Parser, eitherResult, parse, take)
import           Data.Binary.Get            (getWord32le, runGet)
import           Data.ByteString            (ByteString, readFile)
import           Data.ByteString.Char8      (unpack)
import           Data.ByteString.Lazy       (fromStrict)
import           Prelude                    hiding (readFile, take)

-- The DIB section from a bitmap header
data DIB = BM | BA | CI | CP | IC | PT
           deriving (Show, Read)

type Reserved = ByteString

-- The entire bitmap header
data Header = Header DIB Int Reserved Reserved Int
              deriving (Show)
```

Podemos analizar el encabezado de un archivo de mapa de bits fácilmente. Aquí, tenemos 4 funciones de analizador que representan la sección del encabezado de un archivo de mapa de bits:

En primer lugar, la sección DIB se puede leer tomando los primeros 2 bytes

```
dibP :: Parser DIB
dibP = read . unpack <$> take 2
```

Del mismo modo, el tamaño del mapa de bits, las secciones reservadas y el desplazamiento de píxeles también se pueden leer fácilmente:

```
sizeP :: Parser Int
sizeP = fromIntegral . runGet getWord32le . fromStrict <$> take 4

reservedP :: Parser Reserved
reservedP = take 2

addressP :: Parser Int
addressP = fromIntegral . runGet getWord32le . fromStrict <$> take 4
```

que luego se puede combinar en una función de analizador más grande para todo el encabezado:

```
bitmapHeader :: Parser Header
bitmapHeader = do
    dib <- dibP
    sz <- sizeP
    reservedP
    reservedP
    offset <- addressP
    return $ Header dib sz "" "" offset
```

Lea Attoparsec en línea: https://riptutorial.com/es/haskell/topic/9681/attoparsec

Capítulo 8: Bases de datos

Examples

Postgres

Postgresql-simple es una biblioteca de nivel medio de Haskell para comunicarse con una base de datos back-end de PostgreSQL. Es muy simple de usar y proporciona una API segura para escribir y escribir en una base de datos.

Ejecutar una consulta simple es tan fácil como:

```
{-# LANGUAGE OverloadedStrings #-}

import Database.PostgreSQL.Simple

main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  [Only i] <- query_ conn "select 2 + 2"  -- execute with no parameter substitution
  print i
```

Sustitución de parámetros

PostreSQL-Simple admite la sustitución de parámetros para consultas parametrizadas seguras utilizando la `query` :

```
main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  [Only i] <- query conn "select ? + ?" [1, 1]
  print i
```

Ejecución de inserciones o actualizaciones.

Puede ejecutar inserciones / actualizar consultas de SQL usando `execute` :

```
main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  execute conn "insert into people (name, age) values (?, ?)" ["Alex", 31]
```

Lea Bases de datos en línea: https://riptutorial.com/es/haskell/topic/4444/bases-de-datos

Capítulo 9: Bifunctor

Sintaxis

- bimap :: (a -> b) -> (c -> d) -> pac -> pbd
- primero :: (a -> b) -> pac -> pbc
- segundo :: (b -> c) -> pab -> pac

Observaciones

Una ejecución del `Functor` mill es covariante en un *único* parámetro de tipo. Por ejemplo, si `f` es un `Functor` , luego se le asigna un `fa` y una función de la forma `a -> b` , se puede obtener un `fb` (mediante el uso de `fmap`).

Un `Bifunctor` es covariante en *dos* parámetros de tipo. Si `f` es un `Bifunctor` , luego se le asigna un `fab` , y dos funciones, una de `a -> c` , y otra de `b -> d` , entonces se puede obtener un `fcd` (usando `bimap`).

`first` debe pensarse como un `fmap` sobre el primer parámetro de tipo, `second` como un `fmap` sobre el segundo, y `bimap` debe concebirse como un mapeo de dos funciones de forma covariante sobre los parámetros de primer y segundo tipo, respectivamente.

Examples

Instancias comunes de Bifunctor

Tuplas de dos elementos

`(,)` es un ejemplo de un tipo que tiene una instancia de `Bifunctor` .

```
instance Bifunctor (,) where
    bimap f g (x, y) = (f x, g y)
```

`bimap` toma un par de funciones y las aplica a los componentes respectivos de la tupla.

```
bimap (+ 2) (++ "nie") (3, "john") --> (5,"johnnie")
bimap ceiling length (3.5 :: Double, "john" :: String) --> (4,4)
```

Either

`Either` las `Either` instancias de `Bifunctor` selecciona una de las dos funciones para aplicar dependiendo de si el valor es `Left` o `Right` .

```
instance Bifunctor Either where
    bimap f g (Left x) = Left (f x)
    bimap f g (Right y) = Right (g y)
```

primero y segundo

Si se desea mapear de forma covariante solo sobre el primer argumento, o solo sobre el segundo, se debe usar `first` o `second` (en lugar de `bimap`).

```
first :: Bifunctor f => (a -> c) -> f a b -> f c b
first f = bimap f id

second :: Bifunctor f => (b -> d) -> f a b -> f a d
second g = bimap id g
```

Por ejemplo,

```
ghci> second (+ 2) (Right 40)
Right 42
ghci> second (+ 2) (Left "uh oh")
Left "uh oh"
```

Definición de Bifunctor

`Bifunctor` es la clase de tipos con dos parámetros de tipo (`f :: * -> * -> *`), los cuales pueden ser mapeados de forma covariante simultáneamente.

```
class Bifunctor f where
    bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
```

Puede pensarse que `bimap` aplica un par de operaciones `fmap` a un tipo de datos.

Una instancia correcta de `Bifunctor` para un tipo `f` debe satisfacer las *leyes de bifuncores* , que son análogas a las *leyes de functor* :

```
bimap id id = id  -- identity
bimap (f . g) (h . i) = bimap f h . bimap g i  -- composition
```

La clase `Bifunctor` se encuentra en el módulo `Data.Bifunctor` . Para las versiones de GHC> 7.10, este módulo se incluye con el compilador; para versiones anteriores necesita instalar el paquete `bifunctors` .

Lea Bifunctor en línea: https://riptutorial.com/es/haskell/topic/8020/bifunctor

Capítulo 10: Cábala

Sintaxis

- cabal <comando> donde <comando> es uno de:
- **[global]**
  - actualizar
    - Actualiza la lista de paquetes conocidos.
  - instalar
    - Instalar paquetes
  - ayuda
    - Ayuda sobre comandos
  - info
    - Mostrar información detallada sobre un paquete en particular
  - lista
    - Listar paquetes que coincidan con una cadena de búsqueda
  - ha podido recuperar
    - Paquetes de descarga para su posterior instalación.
  - configuración de usuario
    - Visualice y actualice la configuración global del usuario.
- **[paquete]**
  - obtener
    - Descargar / Extraer el código fuente de un paquete (repositorio)
  - en eso
    - Crear un nuevo archivo de paquete .cabal (interactivamente)
  - configurar
    - Prepárese para construir el paquete
  - construir
    - Compilar todos / componentes específicos
  - limpiar
    - Limpiar después de una construcción
  - correr
    - Construye y ejecuta un ejecutable.
  - réplica
    - Abra una sesión de intérprete para el componente dado
  - prueba
    - Ejecutar todas / pruebas específicas en el conjunto de pruebas
  - banco
    - Ejecutar todos / benchmarks específicos
  - comprobar
    - Compruebe el paquete para errores comunes
  - sdist
    - Generar un archivo de distribución de origen (.tar.gz)
  - subir

        - Carga paquetes fuente o documentación a Hackage
    - informe
        - Cargar informes de compilación a un servidor remoto
    - congelar
        - Congelar dependencias
    - límites generales
        - Generar límites de dependencia.
    - eglefino
        - Generar documentación HTML de eglefino
    - hscolor
        - Generar código coloreado HsColour, en formato HTML.
    - dupdo
        - Copie los archivos en las ubicaciones de instalación
    - registro
        - Registrar este paquete con el compilador.
- **[salvadera]**
    - salvadera
        - Crear / modificar / eliminar un arenero
            - cabal sandbox init [BANDERAS]
            - eliminar cuadro de arena cabal [BANDERAS]
            - Cabal sandbox add-source [BANDERAS] CAMINOS
            - cabal sandbox delete-source [FLAGS] Rutas
            - cabal sandbox lista-fuentes [BANDERAS]
            - cabal sandbox hc-pkg [BANDERAS] [-] COMANDO [-] [ARGS]
    - exec
        - Dar un comando de acceso al repositorio de paquetes sandbox
    - réplica
        - Intérprete abierto con acceso a paquetes sandbox.

Examples

Instalar paquetes

Para instalar un nuevo paquete, por ejemplo, aeson:

```
cabal install aeson
```

Trabajando con cajas de arena

Un proyecto de Haskell puede usar los paquetes de todo el sistema o usar un sandbox. Una caja de arena es una base de datos de paquetes aislada y puede evitar conflictos de dependencia, por ejemplo, si varios proyectos de Haskell usan versiones diferentes de un paquete.

Para inicializar un sandbox para un paquete de Haskell, vaya a su directorio y ejecute:

```
cabal sandbox init
```

Ahora los paquetes se pueden instalar simplemente ejecutando `cabal install` .

Listado de paquetes en una caja de arena:

```
cabal sandbox hc-pkg list
```

Eliminar una caja de arena:

```
cabal sandbox delete
```

Añadir dependencia local:

```
cabal sandbox add-source /path/to/dependency
```

Lea Cábala en línea: https://riptutorial.com/es/haskell/topic/4740/cabala

Capítulo 11: Categoria teoria

Examples

La teoría de la categoría como sistema de organización de la abstracción.

La teoría de categorías es una teoría matemática moderna y una rama del álgebra abstracta centrada en la naturaleza de la conexión y la relación. Es útil para proporcionar bases sólidas y lenguaje común a muchas abstracciones de programación altamente reutilizables. Haskell utiliza la teoría de categorías como inspiración para algunas de las clases de tipos principales disponibles tanto en la biblioteca estándar como en varias bibliotecas populares de terceros.

Un ejemplo

La clase de tipos de `Functor` dice que si un tipo `F` crea una instancia de `Functor` (para el cual escribimos el `Functor F`), entonces tenemos una operación genérica

```
fmap :: (a -> b) -> (F a -> F b)
```

lo que nos permite "mapear" sobre `F` . La intuición estándar (pero imperfecta) es que `F a` es un contenedor lleno de valores de tipo `a` y `fmap` nos permite aplicar una transformación a cada uno de estos elementos contenidos. Un ejemplo es `Maybe`

```
instance Functor Maybe where
  fmap f Nothing = Nothing     -- if there are no values contained, do nothing
  fmap f (Just a) = Just (f a) -- else, apply our transformation
```

Dada esta intuición, una pregunta común es "¿por qué no llamar a `Functor` algo obvio como `Mappable` ?".

Un indicio de la teoría de la categoría.

La razón es que Functor se ajusta a un conjunto de estructuras comunes en la teoría de categorías y, por lo tanto, al llamar a `Functor` "Functor" podemos ver cómo se conecta con este cuerpo de conocimiento más profundo.

En particular, la teoría de categorías está muy preocupada por la idea de las flechas de un lugar a otro. En Haskell, el conjunto de flechas más importante son las flechas de función `a -> b` . Una cosa común de estudiar en la teoría de categorías es cómo un conjunto de flechas se relaciona con otro conjunto. En particular, para cualquier constructor de tipo `F` , el conjunto de flechas de la forma `F a -> F b` también son interesantes.

Por lo que un Functor es cualquier `F` tal que existe una conexión entre Haskell normales flechas `a -> b`

y el `F` flechas específicos de `F a -> F b` . La conexión está definida por `fmap` y también reconocemos algunas leyes que deben `fmap`

```
forall (x :: F a) . fmap id x == x

forall (f :: a -> b) (g :: b -> c) . fmap g . fmap f = fmap (g . f)
```

Todas estas leyes surgen naturalmente de la interpretación teórica de la categoría de `Functor` y no serían tan obviamente necesarias si solo `Functor` en `Functor` como relacionado con "mapear sobre elementos".

Definición de una categoría

Una categoría `C` consiste en:

- Una colección de objetos llamados `Obj(C)` ;
- Una colección (llamada `Hom(C)`) de morfismos entre esos objetos. Si `a` y `b` son en `Obj(C)` , entonces un morfismo `f` en `Hom(C)` es típicamente denota `f : a -> b` , y la colección de todos morfismo entre `a` y `b` se denota `hom(a,b)` ;
- Un morfismo especial denominado morfismo de *identidad* : para cada `a : Obj(C)` existe un `id : a -> a` morfismo `id : a -> a` ;
- Un operador de composición (`.`), Tomando dos morfismos `f : a -> b` , `g : b -> c` produciendo un morfismo `a -> c`

que obedecen las siguientes leyes:

```
For all f : a -> x, g : x -> b, then id . f = f and g . id = g
```

```
For all f : a -> b, g : b -> c and h : c -> d, then h . (g . f) = (h . g) . f
```

En otras palabras, la composición con el morfismo de identidad (a la izquierda o a la derecha) no cambia el otro morfismo, y la composición es asociativa.

En Haskell, la `Category` se define como una clase de tipo en Control.Category :

```
-- | A class for categories.
--   id and (.) must form a monoid.
class Category cat where
    -- | the identity morphism
    id :: cat a a

    -- | morphism composition
    (.) :: cat b c -> cat a b -> cat a c
```

En este caso, `cat :: k -> k -> *` objetiva la relación de morfismo: existe un morfismo `cat ab` si y solo si `cat ab` está habitado (es decir, tiene un valor). `a` , `b` y `c` son todos en `Obj(C)` . `Obj(C)` sí está representado por el *tipo* `k` ; por ejemplo, cuando `k ~ *` , como suele ser el caso, los objetos son tipos.

El ejemplo canónico de una categoría en Haskell es la categoría de función:

```
instance Category (->) where
  id = Prelude.id
  (.) = Prelude..
```

Otro ejemplo común es la `Category` de flechas `Kleisli` para una `Monad` :

```
newtype Kleisli m a b = Kleisli (a -> m b)

class Monad m => Category (Kleisli m) where
  id = Kleisli return
  Kleisli f . Kleisli g = Kleisli (f >=> g)
```

Haskell tipifica como categoría

Definición de la categoría

Los tipos Haskell junto con las funciones entre tipos forman (casi †) una categoría. Tenemos un morfismo de identidad (función) (`id :: a -> a`) para cada objeto (tipo) `a` ; y composición de los morfismos (`(.) :: (b -> c) -> (a -> b) -> a -> c`), que obedecen a las leyes de categoría:

```
f . id = f = id . f
h . (g . f) = (h . g) . f
```

Normalmente llamamos a esta categoría **Hask** .

Isomorfismos

En la teoría de categorías, tenemos un isomorfismo cuando tenemos un morfismo que tiene un inverso, en otras palabras, hay un morfismo que se puede componer con él para crear la identidad. En **Hask** esto equivale a tener un par de morfismos `f` , `g` tales que:

```
 f . g == id == g . f
```

Si encontramos un par de tales morfismos entre dos tipos, los llamamos *isomorfos entre sí* .

Un ejemplo de dos tipos isomorfos sería `((),a)` y `a` para algunos `a` . Podemos construir los dos morfismos:

```
f :: ((),a) -> a
f ((),x) = x

g :: a -> ((),a)
g x = ((),x)
```

Y podemos comprobar que `f . g == id == g . f` .

Functores

Un funtor, en la teoría de categorías, va de una categoría a otra, mapeando objetos y morfismos. Estamos trabajando solo en una categoría, la categoría **Hask** of Haskell, por lo que vamos a ver solo los funtores de **Hask** a **Hask** , esos funtores, cuyos orígenes y categorías de destino son los mismos, se denominan **endofunctores** . Nuestros endofunctores serán los tipos polimórficos que toman un tipo y devuelven otro:

```
F :: * -> *
```

Obedecer las leyes de los funtores categóricos (preservar identidades y composición) es equivalente a obedecer las leyes de los funtores de Haskell:

```
fmap (f . g) = (fmap f) . (fmap g)
fmap id = id
```

Entonces, tenemos, por ejemplo, que `[]` , `Maybe` o `(-> r)` son funtores en **Hask** .

Mónadas

Una mónada en la teoría de categorías es un monoide en la **categoría de endofunctores** . Esta categoría tiene endofunctores como objetos `F :: * -> *` y transformaciones naturales (transformaciones entre ellos para todos `forall a . F a -> G a`) como morfismos.

Un objeto monoide se puede definir en una categoría monoidal, y es un tipo que tiene dos morfismos:

```
zero :: () -> M
mappend :: (M,M) -> M
```

Podemos traducir esto aproximadamente a la categoría de endofunctores de Hask como:

```
return :: a -> m a
join :: m (m a) -> m a
```

Y, obedecer las leyes de la mónada es equivalente a obedecer las leyes categóricas de los objetos monoides.

---

† De hecho, la clase de todos los tipos junto con la clase de funciones entre los tipos *no* forman estrictamente una categoría en Haskell, debido a la existencia de `undefined` . Por lo general, esto se soluciona simplemente definiendo los objetos de la categoría **Hask** como tipos sin valores inferiores, que excluyen funciones no terminadas y valores infinitos (codata). Para una discusión detallada de este tema, vea aquí .

Producto de tipos en Hask.

Productos categoricos

En la teoría de categorías, el producto de dos objetos **X** , **Y** es otro objeto **Z** con dos proyecciones: $\pi_1: Z \to X$ y $\pi_2: Z \to Y$; de tal manera que cualquier otro dos morfismos de otro objeto se descomponga de forma única a través de esas proyecciones. En otras palabras, si existe $f_1: W \to X$ y $f_2: W \to Y$, existe un morfismo único $g: W \to Z$ tal que $\pi_1 \circ g = f_1$ y $\pi_2 \circ g = f_2$.

Productos en Hask

Esto se traduce en la categoría **Hask** de tipos de Haskell como sigue, `Z` es producto de `A` , `B` cuando:

```
-- if there are two functions
f1 :: W -> A
f2 :: W -> B
-- we can construct a unique function
g  :: W -> Z
-- and we have two projections
p1 :: Z -> A
p2 :: Z -> B
-- such that the other two functions decompose using g
p1 . g == f1
p2 . g == f2
```

El **tipo de producto de dos tipos** `A` , `B` , que sigue la ley mencionada anteriormente, **es la tupla** de los dos tipos `(A,B)` , y las dos proyecciones son `fst` y `snd` . Podemos verificar que sigue la regla anterior, si tenemos dos funciones `f1 :: W -> A` y `f2 :: W -> B` , podemos descomponerlas de forma única de la siguiente manera:

```
decompose :: (W -> A) -> (W -> B) -> (W -> (A,B))
decompose f1 f2 = (\x -> (f1 x, f2 x))
```

Y podemos comprobar que la descomposición es correcta:

```
fst . (decompose f1 f2) = f1
snd . (decompose f1 f2) = f2
```

Unicidad hasta el isomorfismo

La elección de `(A,B)` como el producto de `A` y `B` no es única. Otra opción lógica y equivalente habría sido:

```
data Pair a b = Pair a b
```

Además, podríamos haber elegido `(B,A)` como el producto, o incluso `(B,A,())` , y podríamos encontrar una función de descomposición como la anterior también siguiendo las reglas:

```
decompose2 :: (W -> A) -> (W -> B) -> (W -> (B,A,()))
decompose2 f1 f2 = (\x -> (f2 x, f1 x, ()))
```

Esto se debe a que el producto no es único, sino *único hasta isomorfismo* . Cada dos productos de `A` y `B` no tienen que ser iguales, pero deben ser isomorfos. Como ejemplo, los dos productos diferentes que acabamos de definir, `(A,B)` y `(B,A,())` , son isomorfos:

```
iso1 :: (A,B) -> (B,A,())
iso1 (x,y) = (y,x,())

iso2 :: (B,A,()) -> (A,B)
iso2 (y,x,()) = (x,y)
```

Singularidad de la descomposición.

Es importante señalar que también la función de descomposición debe ser única. Hay tipos que siguen todas las reglas requeridas para ser producto, pero la descomposición no es única. Como ejemplo, podemos tratar de usar `(A,(B,Bool))` con proyecciones `fst fst . snd` como producto de `A` y `B` :

```
decompose3 :: (W -> A) -> (W -> B) -> (W -> (A,(B,Bool)))
decompose3 f1 f2 = (\x -> (f1 x, (f2 x, True)))
```

Podemos comprobar que funciona:

```
fst         . (decompose3 f1 f2) = f1 x
(fst . snd) . (decompose3 f1 f2) = f2 x
```

Pero el problema aquí es que podríamos haber escrito otra descomposición, a saber:

```
decompose3' :: (W -> A) -> (W -> B) -> (W -> (A,(B,Bool)))
decompose3' f1 f2 = (\x -> (f1 x, (f2 x, False)))
```

Y, como la descomposición **no** es **única** , `(A,(B,Bool))` **no** es el producto de `A` y `B` en **Hask**

Coproducto de tipos en Hask

Intuición

El producto categórico de dos tipos **A** y **B** debe contener la información mínima necesaria para contener dentro de una instancia de tipo **A** o tipo **B.** Podemos ver ahora que el coproducto intuitivo de dos tipos debe ser `Either ab` . Otros candidatos, como `Either a (b,Bool)` , contendrían una parte de información innecesaria y no serían mínimos.

La definición formal se deriva de la definición categórica de coproducto.

Coproductos categóricos

Un coproducto categórico es la noción dual de un producto categórico. Se obtiene directamente invirtiendo todas las flechas en la definición del producto. El coproducto de dos objetos **X** , **Y** es otro objeto **Z** con dos inclusiones: **i_1: X → Z** e **i_2: Y → Z** ; de modo que cualquier otro dos morfismos de **X** e **Y** a otro objeto se descomponen de forma única a través de esas inclusiones. En otras palabras, si hay dos morfismos $\mathbf{f_1: X \to W}$ y $\mathbf{f_2: Y \to W}$, existe un morfismo único **g: Z → W** tal que $\mathbf{g \circ i_1 = f_1}$ y $\mathbf{g \circ i_2 = f_2}$

Coproductos en Hask

La traducción a la categoría de **Hask** es similar a la traducción del producto:

```
-- if there are two functions
f1 :: A -> W
f2 :: B -> W
-- and we have a coproduct with two inclusions
i1 :: A -> Z
i2 :: B -> Z
-- we can construct a unique function
g  :: Z -> W
-- such that the other two functions decompose using g
g . i1 == f1
g . i2 == f2
```

El tipo coproducto de dos tipos `A` y `B` en **Hask** es `Either ab` o cualquier otro tipo isomorfo a la misma:

```
-- Coproduct
-- The two inclusions are Left and Right
data Either a b = Left a | Right b

-- If we have those functions, we can decompose them through the coproduct
decompose :: (A -> W) -> (B -> W) -> (Either A B -> W)
decompose f1 f2 (Left x)  = f1 x
decompose f1 f2 (Right y) = f2 y
```

Haskell Applicative en términos de la teoría de la categoría

El `Functor` de Haskell permite asignar cualquier tipo `a` (un objeto de **Hask**) a un tipo `F a` y también asignar una función `a -> b` (un morfismo de **Hask**) a una función con el tipo `F a -> F b` . Esto corresponde a una definición de teoría de categorías en un sentido en que el funtor conserva la estructura básica de categorías.

Una **categoría monoidal** es una categoría que tiene alguna estructura *adicional* :

- Un producto tensorial (ver Producto de tipos en Hask)
- Una unidad tensorial (objeto unitario)

Tomando un par como nuestro producto, esta definición se puede traducir a Haskell de la siguiente manera:

```
class Functor f => Monoidal f where
    mcat :: f a -> f b -> f (a,b)
    munit :: f ()
```

La clase `Applicative` es equivalente a esta `Monoidal` y, por lo tanto, puede implementarse en términos de esta:

```
instance Monoidal f => Applicative f where
    pure x = fmap (const x) munit
    f <*> fa = (\(f, a) -> f a) <$> (mcat f fa)
```

Lea Categoria teoria en línea: https://riptutorial.com/es/haskell/topic/2261/categoria-teoria

Capítulo 12: Clases de tipo

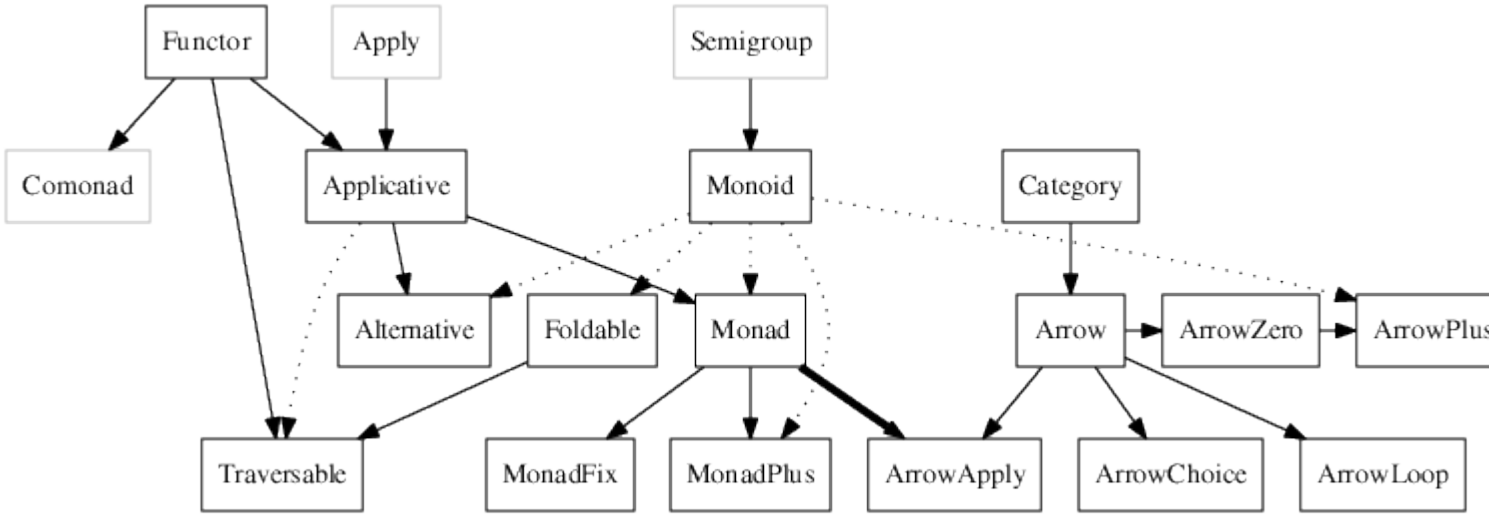
Introducción

Las clases de tipos en Haskell son un medio para definir el comportamiento asociado con un tipo separado de la definición de ese tipo. Mientras que, digamos, en Java, definiría el comportamiento como parte de la definición del tipo, es decir, en una interfaz, clase abstracta o clase concreta, Haskell mantiene estas dos cosas separadas.

Hay una serie de clases de tipos ya definidas en el paquete `base` de Haskell. La relación entre estos se ilustra en la sección de Comentarios a continuación.

Observaciones

El siguiente diagrama tomado del artículo de Typeclassopedia muestra la relación entre las diferentes clases de tipos en Haskell.



Examples

Tal vez y la clase Functor

En Haskell, los tipos de datos pueden tener argumentos como funciones. Tome el tipo `Maybe` por ejemplo.

`Maybe` es un tipo muy útil que nos permite representar la idea de fracaso, o la posibilidad de que sea así. En otras palabras, si existe la posibilidad de que un cálculo falle, usaremos el tipo `Maybe` allí. `Maybe` actúa como una especie de envoltorio para otros tipos, dándoles funcionalidad adicional.

Su declaración actual es bastante simple.

```
Maybe a = Just a | Nothing
```

Lo que esto dice es que un `Maybe` viene en dos formas, un `Just` , que representa el éxito, y una `Nothing` , que representa el fracaso. `Just` toma un argumento que determina el tipo de `Maybe` , y `Nothing` toma ninguno. Por ejemplo, el valor `Just "foo"` tendrá el tipo `Maybe String` , que es un tipo de cadena envuelto con la funcionalidad `Maybe` . El valor `Nothing` tiene tipo `Maybe a` en `a` puede ser de cualquier tipo.

Esta idea de envolver tipos para darles funcionalidad adicional es muy útil y se puede aplicar a más que a `Maybe` . Otros ejemplos incluyen los tipos `Either` , `IO` y lista, cada uno con una funcionalidad diferente. Sin embargo, hay algunas acciones y habilidades que son comunes a todos estos tipos de envoltorios. El más notable de ellos es la capacidad de modificar el valor encapsulado.

Es común pensar en este tipo de tipos como cuadros que pueden tener valores colocados en ellos. Las diferentes cajas tienen diferentes valores y hacen diferentes cosas, pero ninguna es útil sin poder acceder a los contenidos.

Para encapsular esta idea, Haskell viene con una clase de tipos estándar, llamada `Functor` . Se define de la siguiente manera.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Como puede verse, la clase tiene una sola función, `fmap` , de dos argumentos. El primer argumento es una función de un tipo, `a` , a otro, `b` . El segundo argumento es un functor (tipo de envoltura) que contiene un valor de tipo `a` . Devuelve un functor (tipo de envoltorio) que contiene un valor de tipo `b` .

En términos simples, `fmap` toma una función y se aplica al valor dentro de un functor. Es la única función necesaria para que un tipo sea miembro de la clase `Functor` , pero es extremadamente útil. Las funciones que operan en los funtores que tienen aplicaciones más específicas se pueden encontrar en las clases de tipos `Applicative` y `Monad` .

Herencia de clase de tipo: Ord clase de tipo

Haskell soporta una noción de extensión de clase. Por ejemplo, la clase `Ord` hereda todas las operaciones en la `Eq` , pero además tiene una función de `compare` que devuelve un `Ordering` entre valores. `Ord` también puede contener los operadores de comparación de órdenes comunes, así como un método `min` y un método `max` .

La notación `=>` tiene el mismo significado que tiene en la firma de una función y requiere el tipo `a` para implementar la `Eq` , para implementar `Ord` .

```
data Ordering = EQ | LT | GT

class Eq a => Ord a where
    compare :: Ord a => a -> a -> Ordering
    (<)     :: Ord a => a -> a -> Bool
```

```
    (<=)    :: Ord a => a -> a -> Bool
    (>)     :: Ord a => a -> a -> Bool
    (>=)    :: Ord a => a -> a -> Bool
    min     :: Ord a => a -> a -> a
    max     :: Ord a => a -> a -> a
```

Todos los métodos que siguen a la `compare` se pueden derivar de varias formas:

```
x < y   = compare x y == LT
x <= y  = x < y || x == y -- Note the use of (==) inherited from Eq
x > y   = not (x <= y)
x >= y  = not (x < y)

min x y = case compare x y of
               EQ -> x
               LT -> x
               GT -> y

max x y = case compare x y of
               EQ -> x
               LT -> y
               GT -> x
```

Las clases de tipo que a su vez se extienden `Ord` deben implementar al menos el método de `compare` o el método `(<=)` sí mismo, que construye la red de herencia dirigida.

Ecuación

Todos los tipos de datos básicos (como `Int` , `String` , `Eq a => [a]`) de Prelude, excepto las funciones y `IO` tienen instancias de `Eq` . Si un tipo crea una instancia de `Eq` , significa que sabemos cómo comparar dos valores para *valor* o igualdad *estructural* .

```
> 3 == 2
False
> 3 == 3
True
```

Metodos requeridos

- `(==) :: Eq a => a -> a -> Boolean` o `(/=) :: Eq a => a -> a -> Boolean` (si solo se implementa uno, el otro por defecto es la negación del definido uno)

Define

- `(==) :: Eq a => a -> a -> Boolean`
- `(/=) :: Eq a => a -> a -> Boolean`

Superclases directas

Ninguna

Subclases notables

- `Ord`

Ord

Los tipos que crean instancias de `Ord` incluyen, por ejemplo, `Int` , `String` y `[a]` (para los tipos `a` donde hay `Ord a` instancia de `Ord a`). Si un tipo crea una instancia de `Ord` , significa que conocemos un ordenamiento "natural" de valores de ese tipo. Tenga en cuenta que a menudo hay muchas opciones posibles para el ordenamiento "natural" de un tipo y `Ord` nos obliga a favorecerlo.

`Ord` proporciona los operadores estándar `(<=)` , `(<)` , `(>)` , `(>=)` pero los define a todos con un tipo de datos algebraico personalizado

```
data Ordering = LT | EQ | GT

compare :: Ord a => a -> a -> Ordering
```

Metodos requeridos

- `compare :: Ord a => a -> a -> Ordering` o `(<=) :: Ord a => a -> a -> Boolean` (el método de `compare` predeterminado del estándar utiliza `(<=)` en su implementación)

Define

- `compare :: Ord a => a -> a -> Ordering`
- `(<=) :: Ord a => a -> a -> Boolean`
- `(<) :: Ord a => a -> a -> Boolean`
- `(>=) :: Ord a => a -> a -> Boolean`
- `(>) :: Ord a => a -> a -> Boolean`
- `min :: Ord a => a -> a -> a`
- `max :: Ord a => a -> a -> a`

Superclases directas

- `Eq`

Monoide

Los tipos que `Monoid` instancia de `Monoid` incluyen listas, números y funciones con valores de retorno de `Monoid` , entre otros. Para crear una instancia de `Monoid` un tipo debe admitir una

operación binaria asociativa (`mappend` o `(<>)`) que combina sus valores, y tiene un valor especial de "cero" (`mempty`) tal que la combinación de un valor con él no cambie ese valor:

```
mempty  <>  x == x
x <>  mempty  == x

x <> (y <> z) == (x <> y) <> z
```

De manera intuitiva, los tipos `Monoid` son "similares a una lista" en que admiten valores agregados juntos. Alternativamente, los tipos de `Monoid` se pueden considerar como secuencias de valores para los que nos preocupamos por el orden pero no por la agrupación. Por ejemplo, un árbol binario es un `Monoid` , pero al usar las operaciones `Monoid` no podemos presenciar su estructura de bifurcación, solo un recorrido de sus valores (consulte `Foldable` y `Traversable`).

Metodos requeridos

- `mempty :: Monoid m => m`
- `mappend :: Monoid m => m -> m -> m`

Superclases directas

Ninguna

Num

La clase más general para los tipos de números, más precisamente para los anillos , es decir, los números que pueden sumarse y restarse y multiplicarse en el sentido habitual, pero no necesariamente divididos.

Esta clase contiene tanto tipos integrales (`Int` , `Integer` , `Word32` etc.) como tipos fraccionales (`Double` , `Rational` , también números complejos, etc.). En el caso de los tipos finitos, la semántica se entiende generalmente como *aritmética modular* , es decir, con overflow y underflow † .

Tenga en cuenta que las reglas para las clases numéricas están mucho menos obedecidas estrictamente que las leyes de la mónada o monoides, o aquellas para la comparación de la igualdad . En particular, los números de punto flotante generalmente obedecen las leyes solo en un sentido aproximado.

Los métodos

- `fromInteger :: Num a => Integer -> a` . convierta un número entero al tipo de número general (ajuste alrededor del rango, si es necesario). Los literales numéricos de Haskell se pueden entender como un literal monomorfo `Integer` con la conversión general a su alrededor, por lo que puede usar el literal `5` tanto en un contexto `Int` como en un ajuste `Complex Double` .

- `(+) :: Num a => a -> a -> a` . Adición estándar, generalmente entendida como asociativa y

conmutativa, es decir,

```
a + (b + c) ≡ (a + b) + c
a + b ≡ b + a
```

- `(-) :: Num a => a -> a -> a` . Resta, que es la inversa de la suma:

```
(a - b) + b ≡ (a + b) - b ≡ a
```

- `(*) :: Num a => a -> a -> a` . Multiplicación, una operación asociativa que es distributiva sobre la suma:

```
a * (b * c) ≡ (a * b) * c
a * (b + c) ≡ a * b + a * c
```

para las instancias más comunes, la multiplicación también es conmutativa, pero esto definitivamente no es un requisito.

- `negate :: Num a => a -> a` . El nombre completo del operador de negación unario. `-1` es azúcar sintáctico para `negate 1` .

```
-a ≡ negate a ≡ 0 - a
```

- `abs :: Num a => a -> a` . La función de valor absoluto siempre da un resultado no negativo de la misma magnitud

```
abs (-a) ≡ abs a
abs (abs a) ≡ abs a
```

`abs a ≡ 0` solo debería suceder si `a ≡ 0` .

Para los tipos reales , está claro lo que significa no negativo: siempre tienes `abs a >= 0` . Los tipos complejos, etc. no tienen un ordenamiento bien definido, sin embargo, el resultado de los `abs` siempre debe estar en el subconjunto real ‡ (es decir, dar un número que también podría escribirse como un literal de un solo número sin negación).

- `signum :: Num a => a -> a` . La función de signo, de acuerdo con el nombre, produce solo `-1` o `1` , dependiendo del signo del argumento. En realidad, eso solo es cierto para los números reales distintos de cero; en general `signum` se entiende mejor como la función *normalizadora* :

```
abs (signum a) ≡ 1   -- unless a≡0
signum a * abs a ≡ a -- This is required to be true for all Num instances
```

Tenga en cuenta que la sección 6.4.4 del Informe Haskell 2010 requiere explícitamente que esta última igualdad se mantenga para cualquier instancia de `Num` válida.

Algunas bibliotecas, notablemente lineales y hmatrix , tienen una comprensión mucho más laxa de para qué es la clase `Num` : lo tratan como *una forma de sobrecargar a los operadores aritméticos* . Si bien esto es bastante sencillo para `+` y `-` , ya se vuelve problemático con `*` y más con los otros métodos. Por ejemplo, *debería* `*` *significa multiplicación de matrices o multiplicación elemento a elemento?*
Podría decirse que es una mala idea definir tales instancias no numéricas; Por favor considere clases dedicadas como `VectorSpace` .

---

† En particular, los "negativos" de los tipos no firmados se envuelven en grandes positivos, por ejemplo, `(-4 :: Word32) == 4294967292` .

‡ Esto *no se* cumple ampliamente: los tipos de vectores no tienen un subconjunto real. El controvertido `Num` -instances para tales tipos generalmente definen `abs` y `signum` elemento a elemento, que matemáticamente hablando realmente no tiene sentido.

Lea Clases de tipo en línea: https://riptutorial.com/es/haskell/topic/1879/clases-de-tipo

Capítulo 13: Clasificación de los algoritmos

Examples

Tipo de inserción

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x < y     = x:y:ys
                | otherwise = y:(insert x ys)

isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

Ejemplo de uso:

```
> isort [5,4,3,2,1]
```

Resultado:

```
[1,2,3,4,5]
```

Combinar clasificación

Fusión ordenada de dos listas ordenadas.

Preservando los duplicados:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x <= y    = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys
```

Versión de arriba abajo:

```
msort :: Ord a => [a] -> [a]
msort [] = []
msort [a] = [a]
msort xs = merge (msort (firstHalf xs)) (msort (secondHalf xs))

firstHalf  xs = let { n = length xs } in take (div n 2) xs
secondHalf xs = let { n = length xs } in drop (div n 2) xs
```

Se define de esta manera por claridad, no por eficiencia.

Ejemplo de uso:

```
> msort [3,1,4,5,2]
```

Resultado:

```
[1,2,3,4,5]
```

Versión de abajo hacia arriba:

```
msort [] = []
msort xs = go [[x] | x <- xs]
    where
    go [a] = a
    go xs = go (pairs xs)
    pairs (a:b:t) = merge a b : pairs t
    pairs t = t
```

Ordenación rápida

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [a | a <- xs, a < x]
                     ++ [x] ++
               qsort [b | b <- xs, b >= x]
```

Ordenamiento de burbuja

```
bsort :: Ord a => [a] -> [a]
bsort s = case bsort' s of
               t | t == s    -> t
                 | otherwise -> bsort t
  where bsort' (x:x2:xs) | x > x2    = x2:(bsort' (x:xs))
                         | otherwise = x:(bsort' (x2:xs))
        bsort' s = s
```

Orden de permutación

También conocido como bogosort .

```
import Data.List (permutations)

sorted :: Ord a => [a] -> Bool
sorted (x:y:xs) = x <= y && sorted (y:xs)
sorted _        = True

psort :: Ord a => [a] -> [a]
psort = head . filter sorted . permutations
```

Extremadamente ineficiente (en las computadoras de hoy).

Selección de selección

La selección de selección selecciona el elemento mínimo, repetidamente, hasta que la lista esté vacía.

```
import Data.List (minimum, delete)

ssort :: Ord t => [t] -> [t]
ssort [] = []
ssort xs = let { x = minimum xs }
           in  x : ssort (delete x xs)
```

Lea Clasificación de los algoritmos en línea:
https://riptutorial.com/es/haskell/topic/2300/clasificacion-de-los-algoritmos

Capítulo 14: Composición de funciones

Observaciones

El operador de composición de función `(.)` Se define como

```
(.) :: (b -> c) -> (a -> b) ->  (a -> c)
(.)       f           g          x =  f (g x)     -- or, equivalently,

(.)       f           g     =   \x -> f (g x)
(.)       f     =     \g    ->  \x -> f (g x)
(.) =     \f    ->    \g    ->  \x -> f (g x)
(.) =     \f    ->   (\g    -> (\x -> f (g x) ) )
```

El tipo `(b -> c) -> (a -> b) -> (a -> c)` se puede escribir como `(b -> c) -> (a -> b) -> a -> c` porque `->` en las firmas de tipo "asocia" a la derecha, correspondiente a la aplicación de función que asocia a la izquierda,

```
 f g x y z ...    ==    (((f g) x) y) z ...
```

Así que el "flujo de datos" es de derecha a izquierda: `x` "va" a `g` , cuyo resultado va a `f` , produciendo el resultado final:

```
(.)       f           g          x =  r
                                      where r = f (g x)
-- g :: a -> b
-- f ::      b -> c
-- x :: a
-- r ::           c

(.)       f           g     =    q
                                 where q = \x -> f (g x)
-- g :: a -> b
-- f ::      b -> c
-- q :: a      -> c

....
```

Sintácticamente, los siguientes son todos iguales:

```
(.) f g x  =  (f . g) x  =  (f .) g x  =  (. g) f x
```

que es fácil de entender como las "tres reglas de las secciones del operador ", donde el "argumento faltante" solo entra en la ranura vacía cerca del operador:

```
(.) f g    =  (f . g)    =  (f .) g    =  (. g) f
--         1             2             3
```

La `x` , al estar presente en ambos lados de la ecuación, puede omitirse. Esto se conoce como eta-

contracción. Por lo tanto, la forma sencilla de anotar la definición para la composición de la función es simplemente

```
(f . g) x   =   f (g x)
```

Esto, por supuesto, se refiere al "argumento" `x` ; cada vez que escribimos `(f . g)` sin la `x` se conoce como estilo sin puntos.

Examples

Composición de derecha a izquierda

`(.)` nos permite componer dos funciones, alimentando la salida de una como una entrada a la otra:

```
(f . g) x = f (g x)
```

Por ejemplo, si queremos cuadrar el sucesor de un número de entrada, podemos escribir

```
((^2) . succ) 1        --    4
```

También hay `(<<<)` que es un alias para `(.)` . Asi que,

```
(+ 1) <<< sqrt $ 25    --    6
```

Composición de izquierda a derecha

`Control.Category` define `(>>>)` , que, cuando se especializa en funciones, es

```
-- (>>>) :: Category cat => cat a b -> cat b c -> cat a c
-- (>>>) :: (->) a b -> (->) b c -> (->) a c
-- (>>>) :: (a -> b) -> (b -> c) -> (a -> c)
( f >>> g ) x = g (f x)
```

Ejemplo:

```
sqrt >>> (+ 1) $ 25    --    6.0
```

Composición con función binaria.

La composición regular trabaja para funciones únicas. En el caso de binario, podemos definir

```
(f .: g) x y = f (g x y)          -- which is also
             = f ((g x) y)
             = (f . g x) y        -- by definition of (.)
             = (f .) (g x) y
             = ((f .) . g) x y
```

Así, `(f .: g) = ((f .) . g)` por eta-contracción, y además,

```
(.:) f g    = ((f .) . g)
            = (.) (f .) g
            = (.) ((.) f) g
            = ((.) . (.)) f g
```

así que `(.:) = ((.) . (.))` , una definición semi-famosa.

Ejemplos:

```
(map (+1) .: filter) even [1..5]      --  [3,5]
(length   .: filter) even [1..5]      --  2
```

Lea Composición de funciones en línea: https://riptutorial.com/es/haskell/topic/4430/composicion-de-funciones

Capítulo 15: Comprobación rápida

Examples

Declarar una propiedad

En su forma más simple, una *propiedad* es una función que devuelve un `Bool` .

```
prop_reverseDoesNotChangeLength xs = length (reverse xs) == length xs
```

Una propiedad declara un invariante de alto nivel de un programa. El corredor de pruebas QuickCheck evaluará la función con 100 entradas aleatorias y verificará que el resultado sea siempre `True` .

Por convención, las funciones que son propiedades tienen nombres que comienzan con `prop_` .

Comprobando una sola propiedad

La función `quickCheck` prueba una propiedad en 100 entradas aleatorias.

```
ghci> quickCheck prop_reverseDoesNotChangeLength
+++ OK, passed 100 tests.
```

Si una propiedad falla para alguna entrada, `quickCheck` imprime un contraejemplo.

```
prop_reverseIsAlwaysEmpty xs = reverse xs == []  -- plainly not true for all xs

ghci> quickCheck prop_reverseIsAlwaysEmpty
*** Failed! Falsifiable (after 2 tests):
[()]
```

Comprobando todas las propiedades en un archivo

`quickCheckAll` es un ayudante de Haskell de plantillas que encuentra todas las definiciones en el archivo actual cuyo nombre comienza con `prop_` y las prueba.

```
{-# LANGUAGE TemplateHaskell #-}

import Test.QuickCheck (quickCheckAll)
import Data.List (sort)

idempotent :: Eq a => (a -> a) -> a -> Bool
idempotent f x = f (f x) == f x

prop_sortIdempotent = idempotent sort

-- does not begin with prop_, will not be picked up by the test runner
sortDoesNotChangeLength xs = length (sort xs) == length xs
```

```
return []
main = $quickCheckAll
```

Tenga en cuenta que la línea de `return []` es obligatoria. Hace que las definiciones textuales sobre esa línea sean visibles para la Plantilla Haskell.

```
$ runhaskell QuickCheckAllExample.hs
=== prop_sortIdempotent from tree.hs:7 ===
+++ OK, passed 100 tests.
```

Generando datos al azar para tipos personalizados

La clase `Arbitrary` es para tipos que QuickCheck puede generar aleatoriamente.

La implementación mínima de `Arbitrary` es el método `arbitrary` , que se ejecuta en la mónada `Gen` para producir un valor aleatorio.

Aquí hay una instancia de `Arbitrary` para el siguiente tipo de datos de listas no vacías.

```
import Test.QuickCheck.Arbitrary (Arbitrary(..))
import Test.QuickCheck.Gen (oneof)
import Control.Applicative ((<$>), (<*>))

data NonEmpty a = End a | Cons a (NonEmpty a)

instance Arbitrary a => Arbitrary (NonEmpty a) where
    arbitrary = oneof [  -- randomly select one of the cases from the list
        End <$> arbitrary,  -- call a's instance of Arbitrary
        Cons <$>
            arbitrary <*>  -- call a's instance of Arbitrary
            arbitrary  -- recursively call NonEmpty's instance of Arbitrary
        ]
```

Usando la implicación (==>) para verificar las propiedades con condiciones previas

```
prop_evenNumberPlusOneIsOdd :: Integer -> Property
prop_evenNumberPlusOneIsOdd x = even x ==> odd (x + 1)
```

Si desea verificar que una propiedad se cumple dado que se cumple una condición previa, puede usar el operador `==>` . Tenga en cuenta que si es muy poco probable que las entradas arbitrarias coincidan con la condición previa, QuickCheck puede darse por vencido antes.

```
prop_overlySpecific x y = x == 0 ==> x * y == 0

ghci> quickCheck prop_overlySpecific
*** Gave up! Passed only 31 tests.
```

Limitar el tamaño de los datos de prueba

Puede ser difícil probar funciones con poca complejidad asintótica mediante el uso de la comprobación rápida, ya que las entradas aleatorias no suelen tener límites de tamaño. Al agregar un límite superior en el tamaño de la entrada, aún podemos probar estas funciones costosas.

```
import Data.List(permutations)
import Test.QuickCheck

longRunningFunction :: [a] -> Int
longRunningFunction xs = length (permutations xs)

factorial :: Integral a => a -> a
factorial n = product [1..n]

prop_numberOfPermutations xs =
    longRunningFunction xs == factorial (length xs)

ghci> quickCheckWith (stdArgs { maxSize = 10}) prop_numberOfPermutations
```

Al usar `quickCheckWith` con una versión modificada de `stdArgs` , podemos limitar el tamaño de las entradas para que sean como máximo 10. En este caso, ya que estamos generando listas, esto significa que generamos listas de hasta tamaño 10. Nuestra función de permutaciones no tomar demasiado tiempo para estas listas cortas, pero todavía podemos estar razonablemente seguros de que nuestra definición es correcta.

Lea Comprobación rápida en línea: https://riptutorial.com/es/haskell/topic/1156/comprobacion-rapida

Capítulo 16: Concurrencia

Observaciones

Buenos recursos para aprender sobre programación concurrente y paralela en Haskell son:

- Programación paralela y concurrente en Haskell
- la wiki de haskell

Examples

Hilos de desove con `forkIO`

Haskell es compatible con muchas formas de concurrencia y la más obvia es forking un hilo usando `forkIO` .

La función `forkIO :: IO () -> IO ThreadId` realiza una acción `IO` y devuelve su `ThreadId` , mientras que la acción se ejecutará en segundo plano.

Podemos demostrar esto bastante sucintamente usando `ghci` :

```
Prelude Control.Concurrent> forkIO $ (print . sum) [1..100000000]
ThreadId 290
Prelude Control.Concurrent> forkIO $ print "hi!"
"hi!"
-- some time later....
Prelude Control.Concurrent> 50000005000000
```

Ambas acciones se ejecutarán en segundo plano, y la segunda está casi garantizada para finalizar antes de la última.

Comunicando entre hilos con `MVar`

Es muy fácil pasar información entre subprocesos usando el tipo `MVar a` y las funciones que lo acompañan en `Control.Concurrent` :

- `newEmptyMVar :: IO (MVar a)` - crea un nuevo `MVar a`
- `newMVar :: a -> IO (MVar a)` - crea una nueva `MVar` con el valor dado
- `takeMVar :: MVar a -> IO a` - recupera el valor de `MVar` dado, o **bloquea** hasta que haya uno disponible
- `putMVar :: MVar a -> a -> IO ()` - pone el valor dado en el `MVar` , o lo **bloquea** hasta que esté vacío

Sumemos los números de 1 a 100 millones en un hilo y esperemos el resultado:

```
import Control.Concurrent
```

```
main = do
  m <- newEmptyMVar
  forkIO $ putMVar m $ sum [1..10000000]
  print =<< takeMVar m  -- takeMVar will block 'til m is non-empty!
```

Una demostración más compleja podría ser tomar la entrada del usuario y la suma en segundo plano mientras se espera para obtener más información:

```
main2 = loop
  where
    loop = do
        m <- newEmptyMVar
        n <- getLine
        putStrLn "Calculating. Please wait"
        -- In another thread, parse the user input and sum
        forkIO $ putMVar m $ sum [1..(read n :: Int)]
        -- In another thread, wait 'til the sum's complete then print it
        forkIO $ print =<< takeMVar m
        loop
```

Como se indicó anteriormente, si llama a `takeMVar` y la `MVar` está vacía, se bloquea hasta que otro hilo `MVar` algo en la `MVar` , lo que podría resultar en un problema con los filósofos de la cena . Lo mismo ocurre con `putMVar` : si está lleno, se bloqueará hasta que esté vacío.

Tome la siguiente función:

```
concurrent ma mb = do
  a <- takeMVar ma
  b <- takeMVar mb
  putMVar ma a
  putMVar mb b
```

`MVar` las dos funciones con algunos `MVar` s.

```
concurrent ma mb     -- new thread 1
concurrent mb ma     -- new thread 2
```

Lo que podría pasar es que:

1. El hilo 1 lee `ma` y bloquea `ma`
2. El hilo 2 lee `mb` y por lo tanto bloquea `mb`

Ahora, el subproceso 1 no puede leer `mb` porque el subproceso 2 lo ha bloqueado, y el subproceso 2 no puede leer `ma` porque el subproceso 1 lo ha bloqueado. Un callejón sin salida clásico!

Bloques atómicos con software de memoria transaccional

Otra herramienta de concurrencia potente y madura en Haskell es la memoria transaccional de software, que permite que varios subprocesos escriban en una sola variable de tipo `TVar a` de una manera atómica.

`TVar a` es el tipo principal asociado con la mónada `STM` y representa la variable transaccional. Se

usan como `MVar` pero dentro de la mónada `STM` través de las siguientes funciones:

`atomically :: STM a -> IO a`

Realizar una serie de acciones de STM de forma atómica.

`readTVar :: TVar a -> STM a`

Lea el valor de `TVar` , por ejemplo:

```
value <- readTVar t
```

`writeTVar :: TVar a -> a -> STM ()`

Escribe un valor a la `TVar` dada.

```
t <- newTVar Nothing
writeTVar t (Just "Hello")
```

Este ejemplo está tomado de la Wiki de Haskell:

```
import Control.Monad
import Control.Concurrent
import Control.Concurrent.STM

main = do
  -- Initialise a new TVar
  shared <- atomically $ newTVar 0
  -- Read the value
  before <- atomRead shared
  putStrLn $ "Before: " ++ show before
  forkIO $ 25 `timesDo` (dispVar shared >> milliSleep 20)
  forkIO $ 10 `timesDo` (appV ((+) 2) shared >> milliSleep 50)
  forkIO $ 20 `timesDo` (appV pred shared >> milliSleep 25)
  milliSleep 800
  after <- atomRead shared
  putStrLn $ "After: " ++ show after
  where timesDo = replicateM_
        milliSleep = threadDelay . (*) 1000

atomRead = atomically . readTVar
dispVar x = atomRead x >>= print
appV fn x = atomically $ readTVar x >>= writeTVar x . fn
```

Lea Concurrencia en línea: https://riptutorial.com/es/haskell/topic/4426/concurrencia

Capítulo 17: Contenedores - Data.Map

Examples

Construyendo

Podemos crear un Mapa a partir de una lista de tuplas como esta:

```
Map.fromList [("Alex", 31), ("Bob", 22)]
```

Un mapa también se puede construir con un solo valor:

```
> Map.singleton "Alex" 31
fromList [("Alex",31)]
```

También está la función `empty` .

```
empty :: Map k a
```

Data.Map también admite operaciones de conjuntos típicas como `union` , `difference` e `intersection` .

Comprobando si está vacío

Usamos la función `null` para verificar si un mapa dado está vacío:

```
> Map.null $ Map.fromList [("Alex", 31), ("Bob", 22)]
False

> Map.null $ Map.empty
True
```

Encontrar valores

Hay muchas operaciones de consulta en los mapas.

`member :: Ord k => k -> Map ka -> Bool` produce `True` si la clave de tipo `k` está en `Map ka` :

```
> Map.member "Alex" $ Map.singleton "Alex" 31
True
> Map.member "Jenny" $ Map.empty
False
```

`notMember` es similar:

```
> Map.notMember "Alex" $ Map.singleton "Alex" 31
False
```

```
> Map.notMember "Jenny" $ Map.empty
True
```

También puede usar `findWithDefault :: Ord k => a -> k -> Map ka -> a` para obtener un valor predeterminado si la clave no está presente:

```
Map.findWithDefault 'x' 1 (fromList [(5,'a'), (3,'b')]) == 'x'
Map.findWithDefault 'x' 5 (fromList [(5,'a'), (3,'b')]) == 'a'
```

Insertando Elementos

Insertar elementos es simple:

```
> let m = Map.singleton "Alex" 31
fromList [("Alex",31)]

> Map.insert "Bob" 99 m
fromList [("Alex",31),("Bob",99)]
```

Borrando elementos

```
> let m = Map.fromList [("Alex", 31), ("Bob", 99)]
fromList [("Alex",31),("Bob",99)]

> Map.delete "Bob" m
fromList [("Alex",31)]
```

Importando el Módulo

El módulo `Data.Map` en el paquete de `containers` proporciona una estructura de `Map` que tiene implementaciones estrictas y perezosas.

Cuando se usa `Data.Map` , generalmente se importa para evitar choques con funciones ya definidas en Prelude:

```
import qualified Data.Map as Map
```

Así que nos gustaría a continuación, anteponer `Map` función de llamadas con `Map.` , p.ej

```
Map.empty -- give me an empty Map
```

Instancia de monoide

`Map kv` proporciona una instancia de Monoid con la siguiente semántica:

- `mempty` es el `Map` vacío, es decir, igual que `Map.empty`
- `m1 <> m2` es la unión sesgada a la izquierda de `m1` y `m2` , es decir, si alguna tecla está presente tanto en `m1` como en `m2` , entonces el valor de `m1` se selecciona para `m1 <> m2` . Esta

operación también está disponible fuera de la `Monoid` ejemplo como `Map.union` .

Lea Contenedores - Data.Map en línea: https://riptutorial.com/es/haskell/topic/4591/contenedores--data-map

Capítulo 18: Creación de tipos de datos personalizados

Examples

Creando un tipo de datos simple

La forma más fácil de crear un tipo de datos personalizado en Haskell es usar la palabra clave `data` :

```
data Foo = Bar | Biz
```

El nombre del tipo se especifica entre `data` y `=` , y se denomina **constructor de tipo** . Después `=` especificamos todos los **constructores** de **valor** de nuestro tipo de datos, delimitados por `|` firmar. Hay una regla en Haskell según la cual todos los constructores de tipos y valores deben comenzar con una letra mayúscula. La declaración anterior se puede leer como sigue:

> Defina un tipo llamado `Foo` , que tiene dos valores posibles: `Bar` y `Biz` .

Creando variables de nuestro tipo personalizado.

```
let x = Bar
```

La declaración anterior crea una variable llamada `x` de tipo `Foo` . Vamos a verificar esto comprobando su tipo.

```
:t x
```

huellas dactilares

```
x :: Foo
```

Creación de un tipo de datos con parámetros de constructor de valor.

Los constructores de valor son funciones que devuelven un valor de un tipo de datos. Debido a esto, al igual que cualquier otra función, pueden tomar uno o más parámetros:

```
data Foo = Bar String Int | Biz String
```

Vamos a comprobar el tipo de constructor del valor `Bar` .

```
:t Bar
```

huellas dactilares

```
Bar :: String -> Int -> Foo
```

Lo que prueba que `Bar` es de hecho una función.

Creando variables de nuestro tipo personalizado.

```
let x = Bar "Hello" 10
let y = Biz "Goodbye"
```

Creación de un tipo de datos con parámetros de tipo

Los constructores de tipos pueden tomar uno o más parámetros de tipo:

```
data Foo a b = Bar a b | Biz a b
```

Los parámetros de tipo en Haskell deben comenzar con una letra minúscula. Nuestro tipo de datos personalizado aún no es un tipo real. Para crear valores de nuestro tipo, debemos sustituir todos los parámetros de tipo con los tipos reales. Debido `a b` pueden ser de cualquier tipo, nuestros constructores de valores son funciones polimórficas.

Creando variables de nuestro tipo personalizado.

```
let x = Bar "Hello" 10      -- x :: Foo [Char] Integer
let y = Biz "Goodbye" 6.0   -- y :: Fractional b => Foo [Char] b
let z = Biz True False      -- z :: Foo Bool Bool
```

Tipo de datos personalizado con parámetros de registro

Supongamos que queremos crear un tipo de datos Persona, que tenga un nombre y apellido, una edad, un número de teléfono, una calle, un código postal y una ciudad.

Podriamos escribir

```
data Person = Person String String Int Int String String String
```

Si queremos obtener el número de teléfono ahora, necesitamos hacer una función

```
getPhone :: Person -> Int
getPhone (Person _ _ _ phone _ _ _) = phone
```

Bueno, esto no es divertido. Podemos hacerlo mejor utilizando parámetros:

```
data Person' = Person' { firstName     :: String
                       , lastName      :: String
```

```
                        , age           :: Int
                        , phone         :: Int
                        , street        :: String
                        , code          :: String
                        , town          :: String }
```

Ahora tenemos el `phone` función donde

```
:t phone
phone :: Person' -> Int
```

Ahora podemos hacer lo que queramos, por ejemplo:

```
printPhone :: Person' -> IO ()
printPhone = putStrLn . show . phone
```

También podemos vincular el número de teléfono por Coincidencia de patrones :

```
getPhone' :: Person' -> Int
getPhone' (Person {phone = p}) = p
```

Para facilitar el uso de los parámetros, consulte RecordWildCards

Lea Creación de tipos de datos personalizados en línea:
https://riptutorial.com/es/haskell/topic/4057/creacion-de-tipos-de-datos-personalizados

Capítulo 19: Data.Aeson - JSON en Haskell

Examples

Codificación y decodificación inteligentes usando genéricos

La forma más fácil y rápida de codificar un tipo de datos Haskell a JSON con Aeson es mediante el uso de genéricos.

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics
import Data.Text
import Data.Aeson
import Data.ByteString.Lazy
```

Primero creamos un tipo de datos Persona:

```
data Person = Person { firstName :: Text
                     , lastName  :: Text
                     , age       :: Int
                     } deriving (Show, Generic)
```

Para utilizar la función de `encode` y `decode` del paquete `Data.Aeson` , necesitamos que `Person` una instancia de `ToJSON` y `FromJSON` . Dado que derivamos `Generic` for `Person` , podemos crear instancias vacías para estas clases. Las definiciones predeterminadas de los métodos se definen en términos de los métodos proporcionados por la clase de tipo `Generic` .

```
instance ToJSON Person
instance FromJSON Person
```

¡Hecho! Para mejorar la velocidad de codificación podemos cambiar ligeramente la instancia de `ToJSON` :

```
instance ToJSON Person where
    toEncoding = genericToEncoding defaultOptions
```

Ahora podemos usar la función de `encode` para convertir `Person` a un Bytestring (perezoso):

```
encodeNewPerson :: Text -> Text -> Int -> ByteString
encodeNewPerson first last age = encode $ Person first last age
```

Y para decodificar solo podemos usar `decode` :

```
> encodeNewPerson "Hans" "Wurst" 30
"{\"lastName\":\"Wurst\",\"age\":30,\"firstName\":\"Hans\"}"
```

```
> decode $ encodeNewPerson "Hans" "Wurst" 30
Just (Person {firstName = "Hans", lastName = "Wurst", age = 30})
```

Una forma rápida de generar un Data.Aeson.Value

```
{-# LANGUAGE OverloadedStrings #-}
module Main where

import Data.Aeson

main :: IO ()
main = do
  let example = Data.Aeson.object [ "key" .= (5 :: Integer), "somethingElse" .= (2 :: Integer)
] :: Value
  print . encode $ example
```

Campos opcionales

A veces, queremos que algunos campos de la cadena JSON sean opcionales. Por ejemplo,

```
data Person = Person { firstName :: Text
                     , lastName  :: Text
                     , age       :: Maybe Int
                     }
```

Esto se puede lograr por

```
import Data.Aeson.TH

$(deriveJSON defaultOptions{omitNothingFields = True} ''Person)
```

Lea Data.Aeson - JSON en Haskell en línea: https://riptutorial.com/es/haskell/topic/4525/data-aeson---json-en-haskell

Capítulo 20: Data.Text

Observaciones

`Text` es una alternativa más eficiente al tipo de `String` estándar de Haskell. `String` se define como una lista enlazada de caracteres en el Preludio estándar, según el Informe Haskell :

```
type String = [Char]
```

`Text` se representa como una matriz empaquetada de caracteres Unicode. Esto es similar a cómo la mayoría de los otros lenguajes de alto nivel representan cadenas, y ofrece una eficiencia de tiempo y espacio mucho mejor que la versión de lista.

`Text` debe ser preferido sobre la `String` para todo el uso de producción. Una excepción notable depende de una biblioteca que tenga una API de `String` , pero incluso en ese caso puede ser beneficioso utilizar el `Text` internamente y convertirlo a una `String` justo antes de interactuar con la biblioteca.

Todos los ejemplos en este tema usan la extensión de lenguaje `OverloadedStrings` .

Examples

Literales de texto

La extensión de lenguaje `OverloadedStrings` permite el uso de literales de cadena normales para representar valores de `Text` .

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "overloaded"
```

Eliminar espacios en blanco

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "\n\r\t   leading and trailing whitespace   \t\r\n"
```

`strip` elimina los espacios en blanco desde el inicio y el final de un valor de `Text` .

```
ghci> T.strip myText
"leading and trailing whitespace"
```

`stripStart` elimina los espacios en blanco solo desde el principio.

```
ghci> T.stripStart myText
"leading and trailing whitespace   \t\r\n"
```

`stripEnd` elimina los espacios en blanco sólo desde el final.

```
ghci> T.stripEnd myText
"\n\r\t   leading and trailing whitespace"
```

`filter` se puede usar para eliminar espacios en blanco u otros caracteres del medio.

```
ghci> T.filter /=' ' "spaces in the middle of a text string"
"spacesinthemiddleofatextstring"
```

Dividir valores de texto

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "mississippi"
```

`splitOn` divide un `Text` en una lista de `Texts` sobre las ocurrencias de una subcadena.

```
ghci> T.splitOn "ss" myText
["mi","i","ippi"]
```

`splitOn` es el inverso de `intercalate` .

```
ghci> intercalate "ss" (splitOn "ss" "mississippi")
"mississippi"
```

`split` divide un valor de `Text` en partes en caracteres que satisfacen un predicado booleano.

```
ghci> T.split (== 'i') myText
["m","ss","ss","pp",""]
```

Codificación y decodificación de texto

Las funciones de codificación y decodificación para una variedad de codificaciones Unicode se pueden encontrar en el módulo `Data.Text.Encoding` .

```
ghci> import Data.Text.Encoding
ghci> decodeUtf8 (encodeUtf8 "my text")
"my text"
```

Tenga en cuenta que `decodeUtf8` lanzará una excepción en la entrada no válida. Si quiere manejar

usted mismo el UTF-8 no válido, use `decodeUtf8With` .

```
ghci> decodeUtf8With (\errorDescription input -> Nothing) messyOutsideData
```

Comprobando si un texto es una subcadena de otro texto

```
ghci> :set -XOverloadedStrings
ghci> import Data.Text as T
```

`isInfixOf :: Text -> Text -> Bool` comprueba si un `Text` está contenido en algún lugar dentro de otro `Text` .

```
ghci> "rum" `T.isInfixOf` "crumble"
True
```

`isPrefixOf :: Text -> Text -> Bool` comprueba si un `Text` aparece al principio de otro `Text` .

```
ghci> "crumb" `T.isPrefixOf` "crumble"
True
```

`isSuffixOf :: Text -> Text -> Bool` comprueba si un `Text` aparece al final de otro `Text` .

```
ghci> "rumble" `T.isSuffixOf` "crumble"
True
```

Texto de indexación

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text

myText = "mississippi"
```

Los caracteres en índices específicos pueden ser devueltos por la función de `index` .

```
ghci> T.index myText 2
's'
```

La función `findIndex` toma una función de tipo `(Char -> Bool)` y Text y devuelve el índice de la primera aparición de una cadena dada o Nothing si no ocurre.

```
ghci> T.findIndex ('s'==) myText
Just 2
ghci> T.findIndex ('c'==) myText
Nothing
```

La función de `count` devuelve el número de veces que se produce un `Text` consulta dentro de otro

`Text` .

```
ghci> count ("miss"::T.Text) myText
1
```

Lea Data.Text en línea: https://riptutorial.com/es/haskell/topic/3406/data-text

Capítulo 21: Declaraciones de fijeza

Sintaxis

1. infix [integer] ops
2. infixl [integer] ops
3. infixr [integer] ops

Parámetros

Componente de declaración	Sentido
`infixr`	el operador es asociativo por derecho
`infixl`	el operador es asociativo a la izquierda
`infix`	el operador no es asociativo
dígito opcional	precedencia de enlace del operador (rango 0 ... 9, por defecto 9)
`op1, ... , opn`	operadores

Observaciones

Para analizar expresiones que involucran operadores y funciones, Haskell usa declaraciones de corrección para averiguar a dónde va el paréntesis. En orden,

1. Envuelve aplicaciones de función en parens
2. utiliza la prioridad de enlace para envolver grupos de términos separados por operadores de la misma precedencia
3. usa la asociatividad de esos operadores para averiguar cómo agregar parens a estos grupos

Observe que asumimos aquí que los operadores en cualquier grupo dado del paso 2 deben tener todos la misma asociatividad. De hecho, Haskell rechazará cualquier programa donde esta condición no se cumpla.

Como ejemplo del algoritmo anterior, podemos pasar por el proceso de agregar paréntesis a `1 + negate 5 * 2 - 3 * 4 ^ 2 ^ 1`.

```
infixl 6 +
infixl 6 -
infixl 7 *
```

```
infixr 8 ^
```

1. `1 + (negate 5) * 2 - 3 * 4 ^ 2 ^ 1`
2. `1 + ((negate 5) * 2) - (3 * (4 ^ 2 ^ 1))`
3. `(1 + ((negate 5) * 2)) - (3 * (4 ^ (2 ^ 1)))`

Más detalles en la sección 4.4.2 del informe Haskell 98 .

Examples

Asociatividad

`infixl` vs `infixr` vs `infix` describe en qué lados se agruparán los parens. Por ejemplo, considere las siguientes declaraciones de arreglo (en base)

```
infixl 6 -
infixr 5 :
infix  4 ==
```

El `infixl` nos dice que `-` ha dejado asociatividad, lo que significa que `1 - 2 - 3 - 4` se analiza como

```
((1 - 2) - 3) - 4
```

El `infixr` nos dice que `:` tiene asociatividad correcta, lo que significa que `1 : 2 : 3 : []` se analiza como

```
1 : (2 : (3 : []))
```

El `infix` nos dice que `==` no se puede usar sin incluir paréntesis, lo que significa que `True == False == True` es un error de sintaxis. Por otro lado, `True == (False == True)` o `(True == False) == True` están bien.

Los operadores sin una declaración de fijación explícita son `infixl 9` .

Precedencia vinculante

El número que sigue a la información de asociatividad describe en qué orden se aplican los operadores. Siempre debe estar entre `0` y `9` inclusive. Esto se conoce comúnmente como la forma en que el operador se une. Por ejemplo, considere las siguientes declaraciones de arreglo (en base)

```
infixl 6 +
infixl 7 *
```

Como `*` tiene una prioridad de enlace más alta que `+` , leemos `1 * 2 + 3` como

```
(1 * 2) + 3
```

En resumen, cuanto más alto sea el número, más cerca estará el operador de "tirar" de los parens a cada lado.

Observaciones

- La aplicación de la función *siempre se* enlaza más alto que los operadores, por lo que `fx `op` gy` debe interpretarse como `(fx)` op `(gy)` sin importar lo que el operador` `op` y su declaración de fijeza.
- Si se omite la prioridad de enlace en una declaración de `infixl *!?` (por ejemplo, ¡tenemos `infixl *!?`) el valor predeterminado es `9` .

Declaraciones de ejemplo

- `infixr 5 ++`
- `infixl 4 <*>, <*, *>, <**>`
- `infixl 8 `shift`, `rotate`, `shiftL`, `shiftR`, `rotateL`, `rotateR``
- `infix 4 ==, /=, <, <=, >=, >`
- `infix ??`

Lea Declaraciones de fijeza en línea: https://riptutorial.com/es/haskell/topic/4691/declaraciones-de-fijeza

Capítulo 22: Desarrollo web

Examples

Servidor

Servant es una biblioteca para declarar API en el nivel de tipo y luego:

- servidores de escritura (esta parte del servidor puede considerarse un framework web),
- obtener funciones de cliente (en haskell),
- generar funciones de cliente para otros lenguajes de programación,
- Genera documentación para tus aplicaciones web.
- y más...

Servant tiene una API concisa pero potente. Una API simple puede escribirse en muy pocas líneas de código:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}

import Data.Text
import Data.Aeson.Types
import GHC.Generics
import Servant.API

data SortBy = Age | Name

data User = User {
  name :: String,
  age :: Int
} deriving (Eq, Show, Generic)

instance ToJSON User  -- automatically convert User to JSON
```

Ahora podemos declarar nuestra API:

```
type UserAPI = "users" :> QueryParam "sortby" SortBy :> Get '[JSON] [User]
```

que se afirma que deseamos exponer `/users` a `GET` peticiones con una consulta de parámetro `sortby` de tipo `SortBy` y JSON retorno de tipo de `User` en la respuesta.

Ahora podemos definir nuestro manejador:

```
-- This is where we'd return our user data, or e.g. do a database lookup
server :: Server UserAPI
server = return [User "Alex" 31]

userAPI :: Proxy UserAPI
userAPI = Proxy
```

```
app1 :: Application
app1 = serve userAPI server
```

Y el método principal que escucha en el puerto `8081` y sirve nuestra API de usuario:

```
main :: IO ()
main = run 8081 app1
```

Tenga en cuenta que Stack tiene una plantilla para generar API básicas en Servant, que es útil para comenzar a trabajar muy rápido.

Yesod

El proyecto de Yesod se puede crear con una `stack new` utilizando las siguientes plantillas:

- `yesod-minimal` . El andamio más simple posible de Yesod.
- `yesod-mongo` . Utiliza MongoDB como motor DB.
- `yesod-mysql` . Utiliza MySQL como motor de base de datos.
- `yesod-postgres` . Utiliza PostgreSQL como motor de base de datos.
- `yesod-postgres-fay` . Utiliza PostgreSQL como motor de base de datos. Utiliza el lenguaje Fay para front-end.
- `yesod-simple` . Plantilla recomendada para usar, si no necesita base de datos.
- `yesod-sqlite` . Utiliza SQlite como motor DB.

`yesod-bin` paquete `yesod-bin` proporciona el ejecutable `yesod` , que se puede usar para ejecutar el servidor de desarrollo. Tenga en cuenta que también puede ejecutar su aplicación directamente, por `yesod` herramienta `yesod` es opcional.

`Application.hs` contiene código que despacha solicitudes entre los manejadores. También establece la base de datos y la configuración de registro, si los usó.

`Foundation.hs` define el tipo de `App` , que puede verse como un entorno para todos los manejadores. Al estar en la mónada `HandlerT` , puede obtener este valor utilizando la función `getYesod` .

`Import.hs` es un módulo que simplemente reexporta cosas comúnmente utilizadas.

`Model.hs` contiene la plantilla Haskell que genera el código y los tipos de datos utilizados para la interacción DB. Presente solo si está utilizando DB.

`config/models` es donde define su esquema de base de datos. Utilizado por `Model.hs` .

`config/routes` define los URI de la aplicación web. Para cada método HTTP de la ruta, deberá crear un controlador llamado `{method}{RouteR}` .

`static/` directory contiene los recursos estáticos del sitio. Estos son compilados en binario por el módulo de `Settings/StaticFiles.hs` .

`templates/` directory contiene plantillas de Shakespeare que se utilizan cuando se atienden

solicitudes.

Finalmente, el controlador `Handler/` directorio contiene módulos que definen los controladores para las rutas.

Cada controlador es una acción de mónada `HandlerT` basada en IO. Puede inspeccionar los parámetros de solicitud, su cuerpo y otra información, realizar consultas al DB con `runDB` , realizar IO arbitrarias y devolver varios tipos de contenido al usuario. Para servir HTML, se `defaultLayout` función `defaultLayout` que permite una composición ordenada de plantillas shakespearianas.

Lea Desarrollo web en línea: https://riptutorial.com/es/haskell/topic/4721/desarrollo-web

Capítulo 23: Esquemas de recursion

Observaciones

Las funciones mencionadas aquí en los ejemplos se definen con distintos grados de abstracción en varios paquetes, por ejemplo, `data-fix` y `recursion-schemes` (más funciones aquí). Puedes ver una lista más completa buscando en Hayoo .

Examples

Puntos fijos

`Fix` toma un tipo de "plantilla" y ata el nudo recursivo, colocando la plantilla en capas como una lasaña.

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

Dentro de una `Fix f` encontramos una capa de la plantilla `f` . Para rellenar `f` parámetro 's, `Fix f` tapones en *sí mismo.* Así que cuando se mira dentro de la plantilla `f` encontrará una ocurrencia recursiva de `Fix f` .

Aquí es cómo un tipo de datos recursivo típico se puede traducir en nuestro marco de plantillas y puntos fijos. Eliminamos las apariciones recursivas del tipo y marcamos sus posiciones usando el parámetro `r` .

```
{-# LANGUAGE DeriveFunctor #-}

-- natural numbers
-- data Nat = Zero | Suc Nat
data NatF r = Zero_ | Suc_ r deriving Functor
type Nat = Fix NatF

zero :: Nat
zero = Fix Zero_
suc :: Nat -> Nat
suc n = Fix (Suc_ n)


-- lists: note the additional type parameter a
-- data List a = Nil | Cons a (List a)
data ListF a r = Nil_ | Cons_ a r deriving Functor
type List a = Fix (ListF a)

nil :: List a
nil = Fix Nil_
cons :: a -> List a -> List a
cons x xs = Fix (Cons_ x xs)


-- binary trees: note two recursive occurrences
```

```
-- data Tree a = Leaf | Node (Tree a) a (Tree a)
data TreeF a r = Leaf_ | Node_ r a r deriving Functor
type Tree a = Fix (TreeF a)

leaf :: Tree a
leaf = Fix Leaf_
node :: Tree a -> a -> Tree a -> Tree a
node l x r = Fix (Node_ l x r)
```

Doblando una estructura de una capa a la vez

Catamorfismos , o *pliegues* , modelo de recursión primitiva. `cata` derriba un punto fijo capa por capa, utilizando una función de *álgebra* (o *función de plegado*) para procesar cada capa. `cata` requiere una instancia de `Functor` para el tipo de plantilla `f` .

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata f = f . fmap (cata f) . unFix

-- list example
foldr :: (a -> b -> b) -> b -> List a -> b
foldr f z = cata alg
    where alg Nil_ = z
          alg (Cons_ x acc) = f x acc
```

Desplegar una estructura de una capa a la vez

Los anamorfismos , o *despliegues* , modelan la primitiva corecursión. `ana` construye un punto fijo capa por capa, utilizando una función de *coalgebra* (o *función de despliegue*) para producir cada nueva capa. `ana` requiere una instancia de `Functor` para el tipo de plantilla `f` .

```
ana :: Functor f => (a -> f a) -> a -> Fix f
ana f = Fix . fmap (ana f) . f

-- list example
unfoldr :: (b -> Maybe (a, b)) -> b -> List a
unfoldr f = ana coalg
    where coalg x = case f x of
                        Nothing -> Nil_
                        Just (x, y) -> Cons_ x y
```

Tenga en cuenta que `ana` y `cata` son *duales* . Los tipos y las implementaciones son imágenes espejo de la otra.

Despliegue y luego plegado, fusionado.

Es común estructurar un programa para construir una estructura de datos y luego colapsarlo en un solo valor. Esto se llama un *hilomorfismo* o *repliegue.* Es posible elidir la estructura intermedia por completo para mejorar la eficiencia.

```
hylo :: Functor f => (a -> f a) -> (f b -> b) -> a -> b
hylo f g = g . fmap (hylo f g) . f  -- no mention of Fix!
```

Derivación:

```
hylo f g = cata g . ana f
         = g . fmap (cata g) . unFix . Fix . fmap (ana f) . f  -- definition of cata and ana
         = g . fmap (cata g) . fmap (ana f) . f  -- unfix . Fix = id
         = g . fmap (cata g . ana f) . f  -- Functor law
         = g . fmap (hylo f g) . f  -- definition of hylo
```

Recursion primitiva

Los paramorfismos modelan la recursión primitiva. En cada iteración del pliegue, la función de plegado recibe el subárbol para su posterior procesamiento.

```
para :: Functor f => (f (Fix f, a) -> a) -> Fix f -> a
para f = f . fmap (\x -> (x, para f x)) . unFix
```

Las `tails` de Prelude se pueden modelar como un paramorfismo.

```
tails :: List a -> List (List a)
tails = para alg
    where alg Nil_ = cons nil nil  -- [[]]
          alg (Cons_ x (xs, xss)) = cons (cons x xs) xss  -- (x:xs):xss
```

Corecursion primitiva

Modelo de *apomorfismos de la* primitiva corecursión. En cada iteración del despliegue, la función de despliegue puede devolver una semilla nueva o un subárbol completo.

```
apo :: Functor f => (a -> f (Either (Fix f) a)) -> a -> Fix f
apo f = Fix . fmap (either id (apo f)) . f
```

Tenga en cuenta que `apo` y `para` son *duales* . Las flechas en el tipo se voltean; la tupla en `para` es dual a la de `Either` in `apo` , y las implementaciones son imágenes espejo de la otra.

Lea Esquemas de recursion en línea: https://riptutorial.com/es/haskell/topic/2984/esquemas-de-recursion

Capítulo 24: Explotación florestal

Introducción

El registro en Haskell se logra generalmente a través de funciones en la mónada `IO` , y así se limita a funciones no puras o "acciones IO".

Hay varias formas de registrar información en un programa Haskell: desde `putStrLn` (o `print`), hasta bibliotecas como hslogger o `Debug.Trace` .

Examples

Registro con hslogger

El módulo `hslogger` proporciona una API similar a la estructura de `logging` de Python, y admite registradores, niveles y redirección jerárquicamente a los manejadores fuera de `stdout` y `stderr` .

De forma predeterminada, todos los mensajes de nivel `WARNING` y superiores se envían a stderr y todos los demás niveles de registro se ignoran.

```
import           System.Log.Logger (Priority (DEBUG), debugM, infoM, setLevel,
                                     updateGlobalLogger, warningM)

main = do
  debugM "MyProgram.main" "This won't be seen"
  infoM "MyProgram.main" "This won't be seen either"
  warningM "MyProgram.main" "This will be seen"
```

Podemos establecer el nivel de un registrador por su nombre usando `updateGlobalLogger` :

```
  updateGlobalLogger "MyProgram.main" (setLevel DEBUG)

  debugM "MyProgram.main" "This will now be seen"
```

Cada registrador tiene un nombre y están ordenados jerárquicamente, por lo que `MyProgram` es un padre de `MyParent.Module` .

Lea Explotación florestal en línea: https://riptutorial.com/es/haskell/topic/9628/explotacion-florestal

Capítulo 25: Extensiones de lenguaje GHC comunes

Observaciones

Estas extensiones de idioma suelen estar disponibles cuando se utiliza el Compilador de Haskell de Glasgow (GHC), ya que no forman parte del Informe de idiomas de Haskell 2010 aprobado. Para usar estas extensiones, uno debe informar al compilador usando una bandera o colocar un programa de `LANGUAGE` antes de la palabra clave del `module` en un archivo. La documentación oficial se puede encontrar en la sección 7 de la guía del usuario de GCH.

El formato del programa `LANGUAGE` es `{-# LANGUAGE ExtensionOne, ExtensionTwo ... #-}` . Ese es el literal `{-#` seguido de `LANGUAGE` seguido por una lista de extensiones separadas por comas, y finalmente el `#-}` cierre ». Se pueden colocar múltiples programas de `LANGUAGE` en un archivo.

Examples

MultiParamTypeClasses

Es una extensión muy común que permite clases de tipo con múltiples parámetros de tipo. Puedes pensar en MPTC como una relación entre tipos.

```
{-# LANGUAGE MultiParamTypeClasses #-}

class Convertable a b where
    convert :: a -> b

instance Convertable Int Float where
    convert i = fromIntegral i
```

El orden de los parámetros importa.

Los MPTC a veces se pueden reemplazar con familias de tipos.

FlexibleInstances

Las instancias regulares requieren:

```
All instance types must be of the form (T a1 ... an)
where a1 ... an are *distinct type variables*,
and each type variable appears at most once in the instance head.
```

Eso significa que, por ejemplo, mientras puede crear una instancia para `[a]` no puede crear una instancia específicamente para `[Int]` . `FlexibleInstances` relaja que:

```
class C a where
```

```
-- works out of the box
instance C [a] where

-- requires FlexibleInstances
instance C [Int] where
```

SobrecargadoStrings

Normalmente, los literales de cadena en Haskell tienen un tipo de `String` (que es un alias de tipo para `[Char]`). Si bien esto no es un problema para programas educativos más pequeños, las aplicaciones del mundo real a menudo requieren un almacenamiento más eficiente, como `Text` o `ByteString` .

`OverloadedStrings` simplemente cambia el tipo de literales a

```
"test" :: Data.String.IsString a => a
```

Permitir que se pasen directamente a las funciones que esperan un tipo de este tipo. Muchas bibliotecas implementan esta interfaz para sus tipos de cadena, incluidos Data.Text y Data.ByteString, que proporcionan ciertas ventajas de tiempo y espacio sobre `[Char]` .

También hay algunos usos únicos de `OverloadedStrings` como los de la biblioteca Postgresql-simple que permite escribir consultas SQL entre comillas dobles como una cadena normal, pero brinda protección contra la concatenación incorrecta, una fuente notoria de ataques de inyección de SQL.

Para crear una instancia de la clase `IsString` , debe `fromString` función `fromString` . Ejemplo † :

```
data Foo = A | B | Other String deriving Show

instance IsString Foo where
  fromString "A" = A
  fromString "B" = B
  fromString xs  = Other xs

tests :: [ Foo ]
tests = [ "A", "B", "Testing" ]
```

---

† Este ejemplo es cortesía de Lyndon Maydwell (`sordina` en GitHub) que se encuentra aquí .

TupleSections

Una extensión sintáctica que permite aplicar el constructor de tuplas (que es un operador) en una sección:

```
(a,b) == (,) a b

-- With TupleSections
(a,b) == (,) a b == (a,) b == (,b) a
```

N-tuplas

También funciona para tuplas con aridad mayor a dos.

```
(,2,) 1 3 == (1,2,3)
```

Cartografía

Esto puede ser útil en otros lugares donde se usan secciones:

```
map (,"tag") [1,2,3] == [(1,"tag"), (2, "tag"), (3, "tag")]
```

El ejemplo anterior sin esta extensión se vería así:

```
map (\a -> (a, "tag")) [1,2,3]
```

UnicodeSyntax

Una extensión que le permite usar caracteres Unicode en lugar de ciertos operadores y nombres integrados.

ASCII	Unicode	Usos)
`::`	::	tiene tipo
`->`	→	Tipos de funciones, lambdas, ramas de `case` , etc.
`=>`	⇒	restricciones de clase
`forall`	∀	polimorfismo explícito
`<-`	←	`do` notación
`*`	★	el tipo (o tipo) de tipos (por ejemplo, `Int :: ★`)
`>-`	⤛	notación `proc` para `Arrows`
`-<`	⤙	notación `proc` para `Arrows`
`>>-`	⤜	notación `proc` para `Arrows`
`-<<`	⤚	notación `proc` para `Arrows`

Por ejemplo:

```
runST :: (forall s. ST s a) -> a
```

se convertiría

```
runST :: (∀ s. ST s a) → a
```

Tenga en cuenta que el ejemplo `*` vs. `★` es ligeramente diferente: ya que `*` no está reservado, `★` también funciona de la misma manera que `*` para la multiplicación, o cualquier otra función llamada `(*)` , y viceversa. Por ejemplo:

```
ghci> 2 ★ 3
6
ghci> let (*) = (+) in 2 ★ 3
5
ghci> let (★) = (-) in 2 * 3
-1
```

Literales binarios

Haskell estándar le permite escribir literales enteros en decimal (sin ningún prefijo), hexadecimal (precedido por `0x` o `0X`) y octal (precedido por `0o` o `0O`). La extensión `BinaryLiterals` agrega la opción de binario (precedida por `0b` o `0B`).

```
0b1111 == 15     -- evaluates to: True
```

Cuantificación existencial

Esta es una extensión de sistema de tipo que permite tipos que se cuantifican existencialmente, o, en otras palabras, tienen variables de tipo que solo se instancian en tiempo de ejecución † .

Un valor de tipo existencial es similar a una referencia de clase base-abstracta en idiomas OO: no sabe el tipo exacto que contiene, pero puede restringir la *clase* de tipos.

```
data S = forall a. Show a => S a
```

o equivalentemente, con sintaxis GADT:

```
{-# LANGUAGE GADTs #-}
data S where
   S :: Show a => a -> S
```

Los tipos existenciales abren la puerta a cosas como contenedores casi heterogéneos: como se dijo anteriormente, en realidad puede haber varios tipos en un valor `S` , pero todos pueden `show` n, por lo tanto, también puede hacer

```
instance Show S where
    show (S a) = show a   -- we rely on (Show a) from the above
```

Ahora podemos crear una colección de tales objetos:

```
ss = [S 5, S "test", S 3.0]
```

Lo que también nos permite utilizar el comportamiento polimórfico:

```
mapM_ print ss
```

Los existenciales pueden ser muy poderosos, pero tenga en cuenta que en Haskell no son necesarios muy a menudo. En el ejemplo anterior, todo lo que puede hacer con la instancia de `Show` es mostrar (¡duh!) Los valores, es decir, crear una representación de cadena. Por lo tanto, todo el tipo `S` contiene exactamente tanta información como la cadena que se obtiene al mostrarla. Por lo tanto, generalmente es mejor simplemente almacenar esa cadena de inmediato, especialmente porque Haskell es perezoso y, por lo tanto, la cadena al principio solo será un thunk no evaluado de todos modos.

Por otro lado, los existenciales causan algunos problemas únicos. Por ejemplo, la forma en que la información de tipo está "oculta" en un existencial. Si coincide con un patrón en un valor de `S` , tendrá el tipo contenido en el alcance (más precisamente, su instancia de `Show`), pero esta información nunca puede escapar de su alcance, que por lo tanto se convierte en una "sociedad secreta": el compilador no permite que *nada se* escape del ámbito, excepto los valores cuyo tipo ya se conoce desde el exterior. Esto puede provocar errores extraños, como `Couldn't match type 'a0' with '()' 'a0' is untouchable` .

---

† Contraste esto con el polimorfismo paramétrico ordinario, que generalmente se resuelve en el momento de la compilación (lo que permite el borrado de todo tipo).

Los tipos existenciales son diferentes de los tipos Rank-N: estas extensiones son, en términos generales, duales entre sí: para usar valores de un tipo existencial, necesita una función polimórfica (posiblemente restringida), como se `show` en el ejemplo. Una función polimórfica se cuantifica *universalmente* , es decir, funciona *para cualquier* tipo en una clase dada, mientras que la cuantificación existencial significa que funciona *para algún* tipo particular que es a priori desconocido. Si tiene una función polimórfica, eso es suficiente, sin embargo, para pasar funciones polimórficas como argumentos, necesita `{-# LANGUAGE Rank2Types #-}` :

```
genShowSs :: (∀ x . Show x => x -> String) -> [S] -> [String]
genShowSs f = map (\(S a) -> f a)
```

LambdaCase

Una extensión sintáctica que te permite escribir `\case` en lugar de `\arg -> case arg of` .

Considere la siguiente definición de función:

```
dayOfTheWeek :: Int -> String
dayOfTheWeek 0 = "Sunday"
```

```
dayOfTheWeek 1 = "Monday"
dayOfTheWeek 2 = "Tuesday"
dayOfTheWeek 3 = "Wednesday"
dayOfTheWeek 4 = "Thursday"
dayOfTheWeek 5 = "Friday"
dayOfTheWeek 6 = "Saturday"
```

Si desea evitar repetir el nombre de la función, puede escribir algo como:

```
dayOfTheWeek :: Int -> String
dayOfTheWeek i = case i of
    0 -> "Sunday"
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
```

Usando la extensión LambdaCase, puedes escribir eso como una expresión de función, sin tener que nombrar el argumento:

```
{-# LANGUAGE LambdaCase #-}

dayOfTheWeek :: Int -> String
dayOfTheWeek = \case
    0 -> "Sunday"
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
```

RankNTypes

Imagina la siguiente situación:

```
foo :: Show a => (a -> String) -> String -> Int -> IO ()
foo show' string int = do
   putStrLn (show' string)
   putStrLn (show' int)
```

Aquí, queremos pasar una función que convierte un valor en una Cadena, aplicar esa función tanto a un parámetro de cadena como a un parámetro int e imprimirlos. En mi mente, no hay razón para que esto falle! Tenemos una función que funciona en ambos tipos de parámetros que estamos pasando.

Desafortunadamente, esto no va a escribir cheque! GHC infiere `a` tipo basado en su primera aparición en el cuerpo de la función. Es decir, en cuanto nos topamos:

```
putStrLn (show' string)
```

GHC inferirá que `show' :: String -> String` , ya que `string` es una `String` . Se procederá a estallar mientras se intenta `show' int` .

`RankNTypes` permite escribir la firma de tipo de la siguiente manera, cuantificando sobre todas las funciones que satisfacen el tipo `show'` :

```
foo :: (forall a. Show a => (a -> String)) -> String -> Int -> IO ()
```

Este es el rango 2 polimorfismo: estamos afirmando que el `show'` función debe funcionar para todos `a` s *dentro de* nuestra función y la implementación anterior ahora trabaja.

La extensión `RankNTypes` permite el anidamiento arbitrario de todos los bloques `forall ...` en firmas de tipo. En otras palabras, permite el polimorfismo de rango N.

Listas sobrecargadas

añadido en GHC 7.8 .

OverloadedLists, similar a OverloadedStrings , permite que los literales de la lista se analicen de la siguiente manera:

```
[]          -- fromListN 0 []
[x]         -- fromListN 1 (x : [])
[x .. ]     -- fromList (enumFrom x)
```

Esto es útil cuando se trata de tipos como `Set` , `Vector` y `Map` s.

```
['0' .. '9']             :: Set Char
[1 .. 10]                :: Vector Int
[("default",0), (k1,v1)] :: Map String Int
['a' .. 'z']             :: Text
```

`IsList` clase `IsList` en `GHC.Exts` está diseñada para usarse con esta extensión.

`IsList` está equipado con una función de tipo, `Item` y tres funciones, `fromList :: [Item l] -> l` , `toList :: l -> [Item l]` y `fromListN :: Int -> [Item l] -> l` donde `fromListN` es opcional. Las implementaciones típicas son:

```
instance IsList [a] where
  type Item [a] = a
  fromList = id
  toList   = id

instance (Ord a) => IsList (Set a) where
  type Item (Set a) = a
  fromList = Set.fromList
  toList   = Set.toList
```

Ejemplos tomados de OverloadedLists - GHC .

Dependencias funcionales

Si tiene una clase de tipo de múltiples parámetros con los argumentos a, b, c y x, esta extensión le permite expresar que el tipo x se puede identificar de forma única a partir de a, b y c:

```
class SomeClass a b c x | a b c -> x where ...
```

Al declarar una instancia de dicha clase, se verificará contra todas las demás instancias para asegurarse de que se mantiene la dependencia funcional, es decir, no existe ninguna otra instancia con el mismo `abc` pero diferente `x` .

Puede especificar múltiples dependencias en una lista separada por comas:

```
class OtherClass a b c d | a b -> c d, a d -> b where ...
```

Por ejemplo en MTL podemos ver:

```
class MonadReader r m| m -> r where ...
instance MonadReader r ((->) r) where ...
```

Ahora, si tiene un valor de tipo `MonadReader a ((->) Foo) => a` , el compilador puede inferir que `a ~ Foo` , ya que el segundo argumento determina completamente el primero, y simplificará el tipo en consecuencia.

La clase `SomeClass` se puede considerar como una función de los argumentos `abc` que dan como resultado `x` . Tales clases pueden usarse para hacer cálculos en el sistema de tipos.

GADTs

Los tipos de datos algebraicos convencionales son paramétricos en sus variables de tipo. Por ejemplo, si definimos un ADT como

```
data Expr a = IntLit Int
            | BoolLit Bool
            | If (Expr Bool) (Expr a) (Expr a)
```

con la esperanza de que esto descartará estáticamente condicionales no bien tipificados, esto no se comportará como se espera ya que el tipo de `IntLit :: Int -> Expr a` se cuantifica `IntLit :: Int -> Expr a` : para *cualquier* elección de `a` , produce un valor de tipo `Expr a` . En particular, para `a ~ Bool` , tenemos `IntLit :: Int -> Expr Bool` , que nos permite construir algo como `If (IntLit 1) e1 e2` que es lo que el tipo del constructor `If` intentaba descartar.

Los tipos de datos algebraicos generalizados nos permiten controlar el tipo resultante de un constructor de datos para que no sean meramente paramétricos. Podemos reescribir nuestro tipo `Expr` como un GADT como este:

```
data Expr a where
  IntLit :: Int -> Expr Int
```

```
  BoolLit :: Bool -> Expr Bool
  If :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Aquí, el tipo del constructor `IntLit` es `Int -> Expr Int` , y así `IntLit 1 :: Expr Bool` no verificará.

La coincidencia de patrones en un valor GADT provoca el refinamiento del tipo del término devuelto. Por ejemplo, es posible escribir un evaluador para `Expr a` como este:

```
crazyEval :: Expr a -> a
crazyEval (IntLit x) =
   -- Here we can use `(+)` because x :: Int
   x + 1
crazyEval (BoolLit b) =
   -- Here we can use `not` because b :: Bool
   not b
crazyEval (If b thn els) =
  -- Because b :: Expr Bool, we can use `crazyEval b :: Bool`.
  -- Also, because thn :: Expr a and els :: Expr a, we can pass either to
  -- the recursive call to `crazyEval` and get an a back
  crazyEval $ if crazyEval b then thn else els
```

Tenga en cuenta que podemos usar `(+)` en las definiciones anteriores porque cuando, por ejemplo, `IntLit x` coincide con el patrón, también aprendemos que `a ~ Int` (y lo mismo para `not` y `if_then_else_` cuando `a ~ Bool`).

ScopedTypeVariables

`ScopedTypeVariables` permite referirse a tipos cuantificados universalmente dentro de una declaración. Para ser más explícitos:

```
import Data.Monoid

foo :: forall a b c. (Monoid b, Monoid c) => (a, b, c) -> (b, c) -> (a, b, c)
foo (a, b, c) (b', c') = (a :: a, b'', c'')
    where (b'', c'') = (b <> b', c <> c') :: (b, c)
```

Lo importante es que podemos utilizar `a` , `b` y `c` para indicar al compilador en subexpresiones de la declaración (la tupla en el `where` cláusula y el primero `a` en el resultado final). En la práctica, `ScopedTypeVariables` ayuda a escribir funciones complejas como una suma de partes, lo que permite al programador agregar firmas de tipo a valores intermedios que no tienen tipos concretos.

Sinónimo de patrones

Los sinónimos de patrón son abstracciones de patrones similares a cómo las funciones son abstracciones de expresiones.

Para este ejemplo, veamos la interfaz que expone `Data.Sequence` y veamos cómo se puede mejorar con los sinónimos de patrón. El tipo `Seq` es un tipo de datos que, internamente, utiliza una representación complicada para lograr una buena complejidad asintótica para varias operaciones, especialmente para O (1) (des) y un (des) snocing.

Pero esta representación es difícil de manejar y algunas de sus invariantes no pueden expresarse en el sistema de tipos de Haskell. Debido a esto, el tipo `Seq` está expuesto a los usuarios como un tipo abstracto, junto con las funciones de constructor y de acceso que conservan invariantes, entre ellas:

```
empty :: Seq a

(<|) :: a -> Seq a -> Seq a
data ViewL a = EmptyL | a :< (Seq a)
viewl :: Seq a -> ViewL a

(|>) :: Seq a -> a -> Seq a
data ViewR a = EmptyR | (Seq a) :> a
viewr :: Seq a -> ViewR a
```

Pero usar esta interfaz puede ser un poco engorroso:

```
uncons :: Seq a -> Maybe (a, Seq a)
uncons xs = case viewl xs of
    x :< xs' -> Just (x, xs')
    EmptyL -> Nothing
```

Podemos usar patrones de vista para limpiarlo un poco:

```
{-# LANGUAGE ViewPatterns #-}

uncons :: Seq a -> Maybe (a, Seq a)
uncons (viewl -> x :< xs) = Just (x, xs)
uncons _ = Nothing
```

Usando la extensión de lenguaje `PatternSynonyms` , podemos proporcionar una interfaz aún más agradable al permitir que la coincidencia de patrones pretenda que tenemos una lista de contras o de snoc:

```
{-# LANGUAGE PatternSynonyms #-}
import Data.Sequence (Seq)
import qualified Data.Sequence as Seq

pattern Empty :: Seq a
pattern Empty <- (Seq.viewl -> Seq.EmptyL)

pattern (:<) :: a -> Seq a -> Seq a
pattern x :< xs <- (Seq.viewl -> x Seq.:< xs)

pattern (:>) :: Seq a -> a -> Seq a
pattern xs :> x <- (Seq.viewr -> xs Seq.:> x)
```

Esto nos permite escribir `uncons` en un estilo muy natural:

```
uncons :: Seq a -> Maybe (a, Seq a)
uncons (x :< xs) = Just (x, xs)
uncons _ = Nothing
```

RecordWildCards

Ver RecordWildCards

Lea Extensiones de lenguaje GHC comunes en línea:
https://riptutorial.com/es/haskell/topic/1274/extensiones-de-lenguaje-ghc-comunes

Capítulo 26: Fecha y hora

Sintaxis

- addDays :: Integer -> Day -> Day
- diffDays :: Day -> Day -> Integer
- fromGregorian :: Integer -> Int -> Int -> Day

  ```
   convert from proleptic Gregorian calendar. First argument is year, second month number (1-12), third day (1-31). Invalid values will be clipped to the correct range, month first, then day.
  ```

- getCurrentTime :: IO UTCTime

Observaciones

El módulo `Data.Time` del paquete de `time` proporciona soporte para recuperar y manipular valores de fecha y hora:

Examples

Encontrar la fecha de hoy

La fecha y la hora actuales se pueden encontrar con `getCurrentTime` :

```
import Data.Time

print =<< getCurrentTime
-- 2016-08-02 12:05:08.937169 UTC
```

Alternativamente, `fromGregorian` devuelve solo la fecha:

```
fromGregorian 1984 11 17  -- yields a Day
```

Sumando, restando y comparando días

Dado un `Day` , podemos realizar comparaciones aritméticas simples, como agregar:

```
import Data.Time

addDays 1 (fromGregorian 2000 1 1)
-- 2000-01-02
addDays 1 (fromGregorian 2000 12 31)
-- 2001-01-01
```

Sustraer:

```
addDays (-1) (fromGregorian 2000 1 1)
-- 1999-12-31

addDays (-1) (fromGregorian 0 1 1)
-- -0001-12-31
-- wat
```

e incluso encontrar la diferencia:

```
diffDays (fromGregorian 2000 12 31) (fromGregorian 2000 1 1)
365
```

Tenga en cuenta que el orden importa:

```
diffDays (fromGregorian 2000 1 1) (fromGregorian 2000 12 31)
-365
```

Lea Fecha y hora en línea: https://riptutorial.com/es/haskell/topic/4950/fecha-y-hora

Capítulo 27: Flechas

Examples

Composiciones de funciones con múltiples canales.

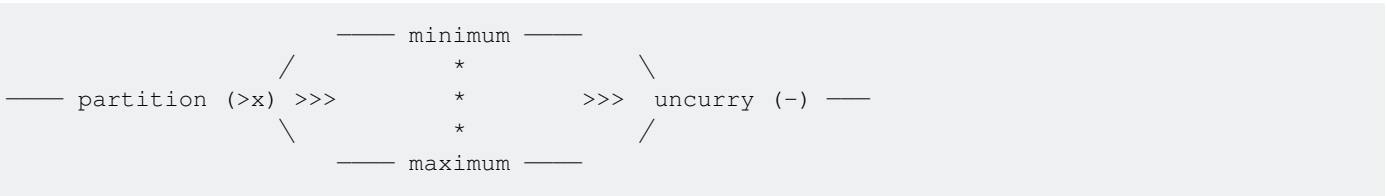
`Arrow` es, vagamente hablando, la clase de morfismos que componen funciones similares, tanto con composición en serie como con "composición paralela". Si bien es más interesante como una *generalización* de funciones, la instancia de `Arrow (->)` sí ya es bastante útil. Por ejemplo, la siguiente función:

```
spaceAround :: Double -> [Double] -> Double
spaceAround x ys = minimum greater - maximum smaller
 where (greater, smaller) = partition (>x) ys
```

También se puede escribir con combinadores de flechas:

```
spaceAround x = partition (>x) >>> minimum *** maximum >>> uncurry (-)
```

Este tipo de composición se puede visualizar mejor con un diagrama:



Aquí,

- El operador `>>>` es solo una versión invertida del ordinario `.` operador de composición (también hay una versión `<<<` que se compone de derecha a izquierda). Canaliza los datos de un paso de procesamiento a otro.
- los `\ / \` indican que el flujo de datos se divide en dos "canales". En términos de tipos de Haskell, esto se realiza con tuplas:

  ```
  partition (>x) :: [Double] -> ([Double], [Double])
  ```

  divide el flujo en dos canales `[Double]` , mientras que

  ```
  uncurry (-) :: (Double,Double) -> Double
  ```

  fusiona dos canales `Double` .
- `***` es el operador de composición † paralelo. Permite que el `maximum` y el `minimum` operen independientemente en diferentes canales de datos. Para funciones, la firma de este operador es

```
(***) :: (b->c) -> (β->γ) -> (b,β)->(c,γ)
```

---

† Al menos en la categoría **Hask** (es decir, en la instancia de `Arrow (->)`), `f***g` no calcula realmente `f` y `g` en paralelo como en, en diferentes subprocesos. Sin embargo, esto sería teóricamente posible.

Lea Flechas en línea: https://riptutorial.com/es/haskell/topic/4912/flechas

Capítulo 28: Función de sintaxis de llamada.

Introducción

La sintaxis de llamada a la función de Haskell, explicada con comparaciones a lenguajes de estilo C, donde corresponda. Esto está dirigido a personas que vienen a Haskell desde un fondo en lenguajes estilo C.

Observaciones

En general, la regla para convertir una llamada de función de estilo C a Haskell, en cualquier contexto (asignación, devolución o incrustada en otra llamada), es reemplazar las comas en la lista de argumentos de estilo C con espacios en blanco y mover la apertura los paréntesis de la llamada de estilo C para contener el nombre de la función y sus parámetros.

Si alguna expresión está envuelta por completo entre paréntesis, estos pares (externos) de paréntesis se pueden eliminar para facilitar la lectura, ya que no afectan el significado de la expresión.
Existen otras circunstancias en las que se pueden eliminar los paréntesis, pero esto solo afecta la legibilidad y la capacidad de mantenimiento.

Examples

Paréntesis en una función básica llamada

Para una llamada de función de estilo C, por ejemplo

```
plus(a, b); // Parentheses surrounding only the arguments, comma separated
```

Entonces el código Haskell equivalente será

```
(plus a b) -- Parentheses surrounding the function and the arguments, no commas
```

En Haskell, los paréntesis no se requieren explícitamente para la aplicación de funciones, y solo se usan para desambiguar expresiones, como en matemáticas; por lo tanto, en los casos en que los corchetes rodean todo el texto de la expresión, los paréntesis en realidad no son necesarios, y lo siguiente también es equivalente:

```
plus a b -- no parentheses are needed here!
```

Es importante recordar que mientras que en los lenguajes de estilo C, la función

Paréntesis en llamadas a funciones incrustadas

En el ejemplo anterior, no terminamos necesitando los paréntesis, porque no afectaron el significado de la declaración. Sin embargo, a menudo son necesarios en expresiones más complejas, como la de abajo.
Cía:

```
plus(a, take(b, c));
```

En Haskell esto se convierte en:

```
(plus a (take b c))
-- or equivalently, omitting the outermost parentheses
plus a (take b c)
```

Tenga en cuenta que esto no es equivalente a:

```
plus a take b c -- Not what we want!
```

Uno podría pensar que debido a que el compilador sabe que `take` es una función, que sería capaz de saber que desea aplicar a los argumentos `b` y `c` , y pasar su resultado a `plus` .
Sin embargo, en Haskell, las funciones a menudo toman otras funciones como argumentos, y se hace poca distinción real entre funciones y otros valores; y así el compilador no puede asumir su intención simplemente porque `take` es una función.

Y así, el último ejemplo es análogo a la siguiente llamada a la función C:

```
plus(a, take, b, c); // Not what we want!
```

Solicitud parcial - Parte 1

En Haskell, las funciones se pueden aplicar parcialmente; podemos pensar que todas las funciones toman un solo argumento y devuelven una función modificada para la cual ese argumento es constante. Para ilustrar esto, podemos agrupar las funciones de la siguiente manera:

```
(((plus) 1) 2)
```

Aquí, la función `(plus)` se aplica a `1` da como resultado la función `((plus) 1)` , que se aplica a `2` , y se obtiene la función `(((plus) 1) 2)` . Debido a que `plus 1 2` es una función que no toma argumentos, puede considerarlo un valor simple; sin embargo, en Haskell, hay poca distinción entre funciones y valores.

Para entrar en más detalles, la función `plus` es una función que agrega sus argumentos.
La función `plus 1` es una función que agrega `1` a su argumento.
La función `plus 1 2` es una función que suma `1` a `2` , que siempre es el valor `3` .

Aplicación parcial - Parte 2

Como otro ejemplo, tenemos el `map` funciones, que toma una función y una lista de valores, y aplica la función a cada valor de la lista:

```
map :: (a -> b) -> [a] -> [b]
```

Digamos que queremos incrementar cada valor en una lista. Puede decidir definir su propia función, que agrega una a su argumento, y `map` esa función a su lista

```
addOne x = plus 1 x
map addOne [1,2,3]
```

pero si tiene otra mirada a la definición de `addOne` , con paréntesis agregados para enfatizar:

```
(addOne) x = ((plus) 1) x
```

La función `addOne` , cuando se aplica a cualquier valor `x` , es la misma que la función parcialmente aplicada `plus 1` aplicada a `x` . Esto significa que las funciones `addOne` y `plus 1` son idénticas, y podemos evitar definir una nueva función simplemente reemplazando `addOne` con `plus 1` , recordando usar paréntesis para aislar `plus 1` como una subexpresión:

```
map (plus 1) [1,2,3]
```

Lea Función de sintaxis de llamada. en línea: https://riptutorial.com/es/haskell/topic/9615/funcion-de-sintaxis-de-llamada-

Capítulo 29: Funciones de orden superior

Observaciones

Las funciones de orden superior son funciones que toman funciones como parámetros y / o devuelven funciones como valores de retorno.

Examples

Conceptos básicos de las funciones de orden superior

Revise la Solicitud Parcial antes de proceder.

En Haskell, una función que puede tomar otras funciones como argumentos o devolver funciones se denomina *función de orden superior* .

Las siguientes son todas *las funciones de orden superior* :

```
map       :: (a -> b) -> [a] -> [b]
filter    :: (a -> Bool) -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
iterate   :: (a -> a) -> a -> [a]
zipWith   :: (a -> b -> c) -> [a] -> [b] -> [c]
scanr     :: (a -> b -> b) -> b -> [a] -> [b]
scanl     :: (b -> a -> b) -> b -> [a] -> [b]
```

Estos son particularmente útiles porque nos permiten crear nuevas funciones además de las que ya tenemos, pasando las funciones como argumentos a otras funciones. De ahí el nombre, *funciones de orden superior* .

Considerar:

```
Prelude> :t (map (+3))
(map (+3)) :: Num b => [b] -> [b]

Prelude> :t (map (=='c'))
(map (=='c')) :: [Char] -> [Bool]

Prelude> :t (map zipWith)
(map zipWith) :: [a -> b -> c] -> [[a] -> [b] -> [c]]
```

Esta capacidad para crear funciones fácilmente (como, por ejemplo, mediante una aplicación parcial como se usa aquí) es una de las características que hace que la programación funcional sea particularmente poderosa y nos permite obtener soluciones cortas y elegantes que de otra forma tomarían docenas de líneas en otros idiomas. Por ejemplo, la siguiente función nos da el número de elementos alineados en dos listas.

```
aligned :: [a] ->  [a] -> Int
aligned xs ys = length (filter id (zipWith (==) xs ys))
```

Expresiones lambda

Las expresiones Lambda son similares a *las funciones anónimas* en otros idiomas.

Las expresiones Lambda son fórmulas abiertas que también especifican variables que deben vincularse. La evaluación (encontrar el valor de una llamada de función) se logra sustituyendo las variables enlazadas en el cuerpo de la expresión lambda, con los argumentos proporcionados por el usuario. En pocas palabras, las expresiones lambda nos permiten expresar funciones mediante la vinculación y sustitución de variables.

Las expresiones Lambda se ven como

```
\x -> let {y = ...x...} in y
```

Dentro de una expresión lambda, las variables en el lado izquierdo de la flecha se consideran unidas en el lado derecho, es decir, el cuerpo de la función.

Considera la función matemática.

```
f(x) = x^2
```

Como una definición de Haskell es

```
f    x =  x^2

f = \x -> x^2
```

lo que significa que la función `f` es equivalente a la expresión lambda `\x -> x^2` .

Considere el parámetro del `map` funciones de orden superior, que es una función de tipo `a -> b` . En caso de que se use solo una vez en una llamada al `map` y en ninguna otra parte del programa, es conveniente especificarla como una expresión lambda en lugar de nombrar dicha función desechable. Escrito como una expresión lambda,

```
\x -> let {y = ...x...} in y
```

`x` tiene un valor de tipo `a` , `...x...` es una expresión de Haskell que se refiere a la variable `x` , y `y` tiene un valor de tipo `b` . Así, por ejemplo, podríamos escribir lo siguiente

```
map (\x -> x + 3)

map (\(x,y) -> x * y)

map (\xs -> 'c':xs) ["apples", "oranges", "mangos"]

map (\f -> zipWith f [1..5] [1..5]) [(+), (*), (-)]
```

Zurra

En Haskell, todas las funciones se consideran como curry: es decir, todas las funciones en Haskell toman solo *un* argumento.

Tomemos la función `div` :

```
div :: Int -> Int -> Int
```

Si llamamos a esta función con 6 y 2, no es sorprendente que obtengamos 3:

```
Prelude> div 6 2
3
```

Sin embargo, esto no se comporta del modo que podríamos pensar. El primer `div 6` se evalúa y **devuelve una función** de tipo `Int -> Int` . Esta función resultante se aplica entonces al valor 2 que produce 3.

Cuando observamos la firma de tipo de una función, podemos cambiar nuestro pensamiento de "toma dos argumentos del tipo `Int` " a "toma un `Int` y devuelve una función que toma un `Int` ". Esto se reafirma si consideramos que las flechas en la notación de tipo están asociadas *a la derecha* , por lo que `div` puede leerse así:

```
div :: Int -> (Int -> Int)
```

En general, la mayoría de los programadores pueden ignorar este comportamiento al menos mientras aprenden el idioma. Desde un punto de vista teórico , "las pruebas formales son más fáciles cuando todas las funciones se tratan de manera uniforme (un argumento hacia adentro, un resultado hacia afuera)".

Lea Funciones de orden superior en línea: https://riptutorial.com/es/haskell/topic/4539/funciones-de-orden-superior

Capítulo 30: Functor

Introducción

`Functor` es la clase de tipos `f :: * -> *` que se puede *mapear de forma* covariante. El mapeo de una función sobre una estructura de datos aplica la función a todos los elementos de la estructura sin cambiar la estructura en sí.

Observaciones

Un Functor puede considerarse como un contenedor para algún valor, o un contexto de cómputo. Algunos ejemplos son `Maybe a` o `[a]` . El artículo de Typeclassopedia tiene una buena reseña de los conceptos detrás de los Functors.

Para ser considerado un Functor real, una instancia debe respetar las 2 leyes siguientes:

Identidad

```
fmap id == id
```

Composición

```
fmap (f . g) = (fmap f) . (fmap g)
```

Examples

Instancias comunes de Functor

Tal vez

`Maybe` es un `Functor` contiene un valor posiblemente ausente:

```
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

La instancia de `Functor` de `Maybe` aplica una función a un valor envuelto en un `Just` . Si el cálculo ha fallado anteriormente (por lo que el valor `Maybe` es un `Nothing`), entonces no hay ningún valor para aplicar la función, por lo que `fmap` es un no-op.

```
> fmap (+ 3) (Just 3)
Just 6
> fmap length (Just "mousetrap")
```

```
Just 9
> fmap sqrt Nothing
Nothing
```

Podemos verificar las leyes de los funtores para esta instancia usando el razonamiento ecuacional. Por la ley de identidad,

```
fmap id Nothing
Nothing  -- definition of fmap
id Nothing  -- definition of id

fmap id (Just x)
Just (id x)  -- definition of fmap
Just x  -- definition of id
id (Just x)  -- definition of id
```

Por la ley de composición,

```
(fmap f . fmap g) Nothing
fmap f (fmap g Nothing)  -- definition of (.)
fmap f Nothing  -- definition of fmap
Nothing  -- definition of fmap
fmap (f . g) Nothing  -- because Nothing = fmap f Nothing, for all f

(fmap f . fmap g) (Just x)
fmap f (fmap g (Just x))  -- definition of (.)
fmap f (Just (g x))  -- definition of fmap
Just (f (g x))  -- definition of fmap
Just ((f . g) x)  -- definition of (.)
fmap (f . g) (Just x)  -- definition of fmap
```

Liza

La instancia de lista de `Functor` aplica la función a cada valor de la lista en su lugar.

```
instance Functor [] where
    fmap f [] = []
    fmap f (x:xs) = f x : fmap f xs
```

Esto también podría escribirse como una lista de comprensión: `fmap f xs = [fx | x <- xs]` .

Este ejemplo muestra que `fmap` generaliza el `map` . `map` solo funciona en listas, mientras que `fmap` funciona en un `Functor` arbitrario.

Se puede demostrar que la ley de identidad se cumple por inducción:

```
-- base case
fmap id []
[]  -- definition of fmap
id []  -- definition of id

-- inductive step
```

```
fmap id (x:xs)
id x : fmap id xs  -- definition of fmap
x : fmap id xs  -- definition of id
x : id xs  -- by the inductive hypothesis
x : xs  -- definition of id
id (x : xs)  -- definition of id
```

y similarmente, la ley de composición:

```
-- base case
(fmap f . fmap g) []
fmap f (fmap g [])  -- definition of (.)
fmap f []  -- definition of fmap
[]  -- definition of fmap
fmap (f . g) []  -- because [] = fmap f [], for all f

-- inductive step
(fmap f . fmap g) (x:xs)
fmap f (fmap g (x:xs))  -- definition of (.)
fmap f (g x : fmap g xs)  -- definition of fmap
f (g x) : fmap f (fmap g xs)  -- definition of fmap
(f . g) x : fmap f (fmap g xs)  -- definition of (.)
(f . g) x : fmap (f . g) xs  -- by the inductive hypothesis
fmap (f . g) xs  -- definition of fmap
```

Funciones

No todos los `Functor` parecen un contenedor. La instancia de Funciones de `Functor` aplica una función al valor de retorno de otra función.

```
instance Functor ((->) r) where
    fmap f g = \x -> f (g x)
```

Tenga en cuenta que esta definición es equivalente a `fmap = (.)` . Así `fmap` generaliza la composición de funciones.

Una vez más comprobando la ley de identidad:

```
fmap id g
\x -> id (g x)  -- definition of fmap
\x -> g x  -- definition of id
g  -- eta-reduction
id g  -- definition of id
```

y la ley de composición:

```
(fmap f . fmap g) h
fmap f (fmap g h)  -- definition of (.)
fmap f (\x -> g (h x))  -- definition of fmap
\y -> f ((\x -> g (h x)) y)  -- definition of fmap
\y -> f (g (h y))  -- beta-reduction
\y -> (f . g) (h y)  -- definition of (.)
fmap (f . g) h  -- definition of fmap
```

Definición de clase de Functor y Leyes

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Una forma de verlo es que `fmap` *eleva* una función de valores a una función de valores en un contexto `f` .

Una instancia correcta de `Functor` debería satisfacer las *leyes de* los *funtores* , aunque el compilador no las impone:

```
fmap id = id                    -- identity
fmap f . fmap g = fmap (f . g)  -- composition
```

Hay un alias de infijo de uso `fmap` para `fmap` llamado `<$>` .

```
infixl 4 <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

Reemplazo de todos los elementos de un Functor con un solo valor

El módulo `Data.Functor` contiene dos combinadores, `<$` y `$>` , que ignoran todos los valores contenidos en un functor, reemplazándolos con un único valor constante.

```
infixl 4 <$, $>

<$ :: Functor f => a -> f b -> f a
(<$) = fmap . const

$> :: Functor f => f a -> b -> f b
($>) = flip (<$)
```

`void` ignora el valor de retorno de un cálculo.

```
void :: Functor f => f a -> f ()
void = (() <$)
```

Funtores polinomiales

Hay un conjunto útil de combinadores de tipo para construir grandes `Functor` s a partir de los más pequeños. Estos son instructivos como ejemplos de `Functor` , y también son útiles como una técnica para la programación genérica, porque se pueden usar para representar una gran clase de functores comunes.

El functor de la identidad.

El functor de la identidad simplemente envuelve su argumento. Es una implementación a nivel de tipo del combinador `I` del cálculo de SKI.

```
newtype I a = I a

instance Functor I where
    fmap f (I x) = I (f x)
```

`I` pueden encontrar, bajo el nombre de `Identity` , en el módulo `Data.Functor.Identity` .

El functor constante

El functor constante ignora su segundo argumento, que contiene solo un valor constante. Es un análogo de nivel de tipo de `const` , el combinador `K` del cálculo de SKI.

```
newtype K c a = K c
```

Tenga en cuenta que `K ca` no contiene `a` -valores; `K ()` es isomorfo a `Proxy` . Esto significa que `K` implementación de 's `fmap` no hace ninguna asignación en absoluto!

```
instance Functor (K c) where
    fmap _ (K c) = K c
```

`K` también se conoce como `Const` , de `Data.Functor.Const` .

Los functores restantes en este ejemplo combinan funtores más pequeños en otros más grandes.

Productos funcionales

El producto functor toma un par de functores y los empaca. Es análogo a una tupla, excepto que while `(,) :: * -> * -> *` opera en `types * , (:*:) :: (* -> *) -> (* -> *) -> (* -> *)` opera en los `functors * -> *` .

```
infixl 7 :*:
data (f :*: g) a = f a :*: g a

instance (Functor f, Functor g) => Functor (f :*: g) where
    fmap f (fx :*: gy) = fmap f fx :*: fmap f gy
```

Este tipo se puede encontrar, bajo el nombre `Product` , en el módulo `Data.Functor.Product` .

Functor coproductos

Al igual que `:*:` es análogo a `(,) :+:` es el análogo a nivel de functor de `Either` de los `Either` .

```
infixl 6 :+:
data (f :+: g) a = InL (f a) | InR (g a)

instance (Functor f, Functor g) => Functor (f :+: g) where
    fmap f (InL fx) = InL (fmap f fx)
    fmap f (InR gy) = InR (fmap f gy)
```

`:+:` se puede encontrar bajo el nombre `Sum` , en el módulo `Data.Functor.Sum` .

Composición funcional

Finalmente, `:.:` Funciona como un nivel de tipo `(.)` , Tomando la salida de un functor y conectándola a la entrada de otra.

```
infixr 9 :.:
newtype (f :.: g) a = Cmp (f (g a))

instance (Functor f, Functor g) => Functor (f :.: g) where
    fmap f (Cmp fgx) = Cmp (fmap (fmap f) fgx)
```

El tipo de `Compose` se puede encontrar en `Data.Functor.Compose`

Funtores polinómicos para la programación genérica.

`I` , `K` `:*:` , `:+:` y `:.:` Se puede considerar como un kit de bloques de construcción para una cierta clase de tipos de datos simples. El kit se vuelve especialmente poderoso cuando lo combinas con puntos fijos porque los tipos de datos construidos con estos combinadores son automáticamente instancias de `Functor` . Usted usa el kit para crear un tipo de plantilla, marcando puntos recursivos con `I` , y luego lo conecta a `Fix` para obtener un tipo que se puede usar con el zoológico estándar de esquemas de recursión.

Nombre	Como un tipo de datos	Usando el kit de functor
Pares de valores	`data Pair a = Pair aa`	`type Pair = I :*: I`
Grid de dos por dos	`type Grid a = Pair (Pair a)`	`type Grid = Pair :.: Pair`
Números naturales	`data Nat = Zero | Succ Nat`	`type Nat = Fix (K () :+: I)`
Liza	`data List a = Nil | Cons a (List a)`	`type List a = Fix (K () :+: K a :*: I)`
Árboles binarios	`data Tree a = Leaf | Node (Tree a) a (Tree a)`	`type Tree a = Fix (K () :+: I :*: K a :*: I)`

Nombre	Como un tipo de datos	Usando el kit de functor
Rosales	`data Rose a = Rose a (List (Rose a))`	`type Rose a = Fix (K a :*: List :.: I)`

Este enfoque de "kit" para diseñar tipos de datos es la idea detrás de *las* bibliotecas de *programación genéricas* como `generics-sop` . La idea es escribir operaciones genéricas usando un kit como el que se presentó anteriormente, y luego usar una clase de tipos para convertir tipos de datos arbitrarios desde y hacia su representación genérica:

```
class Generic a where
    type Rep a  -- a generic representation built using a kit
    to :: a -> Rep a
    from :: Rep a -> a
```

Functors en la teoría de categorías

Un Functor se define en la teoría de categorías como un mapa que preserva la estructura (un 'homomorfismo') entre categorías. Específicamente, (todos) los objetos se asignan a los objetos, y (todas) las flechas se asignan a las flechas, de manera que se conservan las leyes de la categoría.

La categoría en la que los objetos son tipos de Haskell y los morfismos en funciones de Haskell se denomina **Hask** . Entonces, un functor de **Hask** a **Hask** consistiría en un mapeo de tipos a tipos y un mapeo de funciones a funciones.

La relación que este concepto teórico de la categoría guarda con la estructura de programación de Haskell, `Functor` es bastante directa. La asignación de tipos a tipos toma la forma de un tipo `f :: * -> *` , y la asignación de funciones a funciones toma la forma de una función `fmap :: (a -> b) -> (fa -> fb)` . Poniéndolos juntos en una clase,

```
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b
```

`fmap` es una operación que toma una función (un tipo de morfismo), `:: a -> b` , y la asigna a otra función, `:: fa -> fb` . Se supone (pero se deja al programador para garantizar) que las instancias de `Functor` son en realidad functors matemáticos, preservando la estructura categórica de **Hask** :

```
fmap (id {- :: a -> a -})  ==  id {- :: f a -> f a -}
fmap (h . g)               ==  fmap h . fmap g
```

`fmap` levanta una función `:: a -> b` en una subcategoría de **Hask** de una manera que preserva tanto la existencia de flechas de identidad como la asociatividad de la composición.

La clase `Functor` solo codifica los functores *endo* en **Hask** . Pero en matemáticas, los funtores pueden mapear entre categorías arbitrarias. Una codificación más fiel de este concepto se vería así:

```
class Category c where
    id  :: c i i
    (.) :: c j k -> c i j -> c i k

class (Category c1, Category c2) => CFunctor c1 c2 f where
    cfmap :: c1 a b -> c2 (f a) (f b)
```

La clase de Functor estándar es un caso especial de esta clase en la que las categorías de origen y destino son **Hask** . Por ejemplo,

```
instance Category (->) where        -- Hask
    id    = \x -> x
    f . g = \x -> f (g x)

instance CFunctor (->) (->) [] where
    cfmap = fmap
```

Functor de derivación

La extensión de lenguaje `DeriveFunctor` permite a GHC generar instancias de `Functor` automáticamente.

```
{-# LANGUAGE DeriveFunctor #-}

data List a = Nil | Cons a (List a) deriving Functor

-- instance Functor List where            -- automatically defined
--   fmap f Nil = Nil
--   fmap f (Cons x xs) = Cons (f x) (fmap f xs)

map :: (a -> b) -> List a -> List b
map = fmap
```

Lea Functor en línea: https://riptutorial.com/es/haskell/topic/3800/functor

Capítulo 31: Functor Aplicativo

Introducción

`Applicative` es la clase de tipos `f :: * -> *` que permite la aplicación de funciones elevadas sobre una estructura donde la función también está incrustada en esa estructura.

Observaciones

Definición

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Tenga en cuenta la restricción de `Functor` en `f` . La función `pure` devuelve su argumento incrustado en la estructura `Applicative` . La función de infijo `<*>` (pronunciada "aplicar") es muy similar a `fmap` excepto con la función incorporada en la estructura `Applicative` .

Una instancia correcta de `Applicative` debería satisfacer las *leyes aplicativas* , aunque estas no son aplicadas por el compilador:

```
pure id <*> a = a                              -- identity
pure (.) <*> a <*> b <*> c = a <*> (b <*> c)   -- composition
pure f <*> pure a = pure (f a)                 -- homomorphism
a <*> pure b = pure ($ b) <*> a                -- interchange
```

Examples

Definición alternativa

Dado que cada `fmap` Functor es un Functor , `fmap` siempre se puede usar en él; Por lo tanto, la esencia de Applicative es el emparejamiento de contenidos transportados, así como la capacidad de crearlo:

```
class Functor f => PairingFunctor f where
  funit :: f ()                  -- create a context, carrying nothing of import
  fpair :: (f a,f b) -> f (a,b)  -- collapse a pair of contexts into a pair-carrying context
```

Esta clase es isomorfa al `Applicative` .

```
pure a = const a <$> funit = a <$ funit

fa <*> fb = (\(a,b) -> a b) <$> fpair (fa, fb) = uncurry ($) <$> fpair (fa, fb)
```

A la inversa,

```
funit = pure ()

fpair (fa, fb) = (,) <$> fa <*> fb
```

Ejemplos comunes de aplicativo

Tal vez

`Maybe` sea un funtor aplicativo que contenga un valor posiblemente ausente.

```
instance Applicative Maybe where
    pure = Just

    Just f <*> Just x = Just $ f x
    _ <*> _ = Nothing
```

`pure` levanta el valor dado en `Maybe` aplicando `Just` a él. La función `(<*>)` aplica una función envuelta en un `Maybe` a un valor en `Maybe` . Si tanto la función como el valor están presentes (construidos con `Just`), la función se aplica al valor y se devuelve el resultado final. Si falta alguno, el cálculo no puede continuar y, en su lugar, `Nothing` se devuelve `Nothing` .

Liza

Una forma para que las listas se ajusten a la firma de tipo `<*> :: [a -> b] -> [a] -> [b]` es tomar el producto cartesiano de las dos listas, emparejando cada elemento de la primera lista con cada uno Elemento del segundo:

```
fs <*> xs = [f x | f <- fs, x <- xs]
          -- = do { f <- fs; x <- xs; return (f x) }

pure x = [x]
```

Esto generalmente se interpreta como emulando el no determinismo, con una lista de valores representando un valor no determinista cuyos valores posibles se extienden sobre esa lista; por lo tanto, una combinación de dos valores no deterministas abarca todas las combinaciones posibles de los valores en las dos listas:

```
ghci> [(+1),(+2)] <*> [3,30,300]
[4,31,301,5,32,302]
```

Flujos infinitos y listas zip

Hay una clase de `Applicative` que "comprimen" sus dos entradas juntas. Un ejemplo simple es el

de las corrientes infinitas:

```
data Stream a = Stream { headS :: a, tailS :: Stream a }
```

La instancia `Applicative Stream` aplica un flujo de funciones a un flujo de argumentos puntuales, emparejando los valores en los dos flujos por posición. `pure` devuelve una secuencia constante, una lista infinita de un solo valor fijo:

```
instance Applicative Stream where
    pure x = let s = Stream x s in s
    Stream f fs <*> Stream x xs = Stream (f x) (fs <*> xs)
```

Las listas también admiten una instancia `Applicative` "zippy", para la que existe el `ZipList` ZipList:

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
    ZipList xs <*> ZipList ys = ZipList $ zipWith ($) xs ys
```

Dado que `zip` recorta su resultado de acuerdo con la información más corta, la única implementación de `pure` que cumple con las leyes de `Applicative` es una que devuelve una lista infinita:

```
    pure a = ZipList (repeat a)   -- ZipList (fix (a:)) = ZipList [a,a,a,a,...
```

Por ejemplo:

```
ghci> getZipList $ ZipList [(+1),(+2)] <*> ZipList [3,30,300]
[4,32]
```

Las dos posibilidades nos recuerdan el producto externo y el interno, similar a multiplicar una matriz de 1 columna (`nx 1`) con una de 1 fila (`1 xm`) en el primer caso, obteniendo la matriz `nxm` como resultado (pero aplanada)); o multiplicar las matrices de 1 fila y 1 columna (pero sin resumir) en el segundo caso.

Funciones

Cuando se especializa en funciones `(->) r` , las firmas de tipo `pure` y `<*>` coinciden con las de los combinadores `K` y `S` , respectivamente:

```
pure :: a -> (r -> a)
<*> :: (r -> (a -> b)) -> (r -> a) -> (r -> b)
```

`pure` debe ser `const` , y `<*>` toma un par de funciones y las aplica a un argumento fijo, aplicando los dos resultados:

```
instance Applicative ((->) r) where
```

```
    pure = const
    f <*> g = \x -> f x (g x)
```

Las funciones son el prototípico "zippy" aplicativo. Por ejemplo, dado que las corrientes infinitas son isomorfas para `(->) Nat` , ...

```
-- | Index into a stream
to :: Stream a -> (Nat -> a)
to (Stream x xs) Zero = x
to (Stream x xs) (Suc n) = to xs n

-- | List all the return values of the function in order
from :: (Nat -> a) -> Stream a
from f = from' Zero
    where from' n = Stream (f n) (from' (Suc n))
```

... la representación de flujos en una orden superior produce automáticamente la instancia de Zippy `Applicative` .

Lea Functor Aplicativo en línea: https://riptutorial.com/es/haskell/topic/8162/functor-aplicativo

Capítulo 32: GHCJS

Introducción

GHCJS es un compilador de Haskell a JavaScript que utiliza la API de GHC.

Examples

Ejecutando "¡Hola mundo!" con Node.js

`ghcjs` se puede invocar con los mismos argumentos de línea de comando que `ghc` . Los programas generados se pueden ejecutar directamente desde el shell con Node.js y SpiderMonkey jsshell . por ejemplo:

```
$ ghcjs -o helloWorld helloWorld.hs
$ node helloWorld.jsexe/all.js
Hello world!
```

Lea GHCJS en línea: https://riptutorial.com/es/haskell/topic/9260/ghcjs

Capítulo 33: Google Protocol Buffers

Observaciones

Para usar Protocol Buffers con Haskell debes instalar el paquete `htprotoc` :

1. Clona el proyecto desde Github.
2. Usa Stack para construir e instalar

Ahora debería encontrar el ejecutable `hprotoc` en `$HOME/.local/bin/` .

Examples

Creando, construyendo y usando un simple archivo .proto

Primero vamos a crear un sencillo `.proto` archivo `person.proto`

```
package Protocol;

message Person {
    required string firstName = 1;
    required string lastName  = 2;
    optional int32  age       = 3;
}
```

Después de guardar, ahora podemos crear los archivos Haskell que podemos usar en nuestro proyecto ejecutando

```
$HOME/.local/bin/hprotoc --proto_path=. --haskell_out=. person.proto
```

Deberíamos obtener una salida similar a esta:

```
Loading filepath: "/<path-to-project>/person.proto"
All proto files loaded
Haskell name mangling done
Recursive modules resolved
./Protocol/Person.hs
./Protocol.hs
Processing complete, have a nice day.
```

`hprotoc` creará una nueva carpeta `Protocol` en el directorio actual con `Person.hs` que simplemente podemos importar en nuestro proyecto de haskell:

```
import Protocol (Person)
```

Como siguiente paso, si usa Stack, agregue

```
    protocol-buffers
  , protocol-buffers-descriptor
```

para `build-depends:` y

```
Protocol
```

a `exposed-modules` en su archivo `.cabal` .

Si recibimos ahora un mensaje entrante de una transmisión, el mensaje tendrá el tipo `ByteString` .

Para transformar el `ByteString` (que obviamente debería contener datos de "Persona" codificados) en nuestro tipo de datos Haskell, debemos llamar a la función `messageGet` que importamos por

```
import Text.ProtocolBuffers (messageGet)
```

que permite crear un valor de tipo `Person` utilizando:

```
transformRawPerson :: ByteString -> Maybe Person
transformRawPerson raw = case messageGet raw of
    Left   _           -> Nothing
    Right (person, _)  -> Just person
```

Lea Google Protocol Buffers en línea: https://riptutorial.com/es/haskell/topic/5018/google-protocol-buffers

Capítulo 34: Gráficos con brillo

Examples

Instalar Gloss

El brillo se instala fácilmente con la herramienta Cabal. Habiendo instalado Cabal, uno puede ejecutar Cabal `cabal install gloss` para instalar Gloss.

Alternativamente, el paquete puede construirse desde la fuente, descargando la fuente desde Hackage o GitHub , y haciendo lo siguiente:

1. Ingrese al directorio `gloss/gloss-rendering/` e `cabal install`
2. Ingrese al directorio `gloss/gloss/` y una vez más haga la `cabal install`

Consiguiendo algo en la pantalla

En Gloss, se puede usar la función de `display` para crear gráficos estáticos muy simples.

Para usar este se necesita primero `import Graphics.Gloss` . Luego en el código hay que hacer lo siguiente:

```
main :: IO ()
main = display window background drawing
```

`window` es de tipo `Display` que se puede construir de dos maneras:

```
-- Defines window as an actual window with a given name and size
window = InWindow name (width, height) (0,0)

-- Defines window as a fullscreen window
window = FullScreen
```

Aquí el último argumento `(0,0)` en `InWindow` marca la ubicación de la esquina superior izquierda.

Para versiones anteriores a la 1.11: en versiones anteriores de Gloss `FullScreen` toma otro argumento que debe ser el tamaño del marco que se dibuja y que a su vez se estira al tamaño de pantalla completa, por ejemplo: `FullScreen (1024,768)`

`background` es de tipo `Color` . Define el color de fondo, por lo que es tan simple como:

```
background = white
```

Entonces llegamos al dibujo en sí. Los dibujos pueden ser muy complejos. La forma de especificarlos se tratará en otra parte ([uno puede referirse a esto por el momento] [1]), pero puede ser tan simple como el siguiente círculo con un radio de 80:

```
drawing = Circle 80
```

Ejemplo de resumen

Como se indica más o menos en la documentación de Hackage, obtener algo en la pantalla es tan fácil como:

```
import Graphics.Gloss

main :: IO ()
main = display window background drawing
    where
      window = InWindow "Nice Window" (200, 200) (0, 0)
      background = white
      drawing = Circle 80
```

Lea Gráficos con brillo en línea: https://riptutorial.com/es/haskell/topic/5570/graficos-con-brillo

Capítulo 35: Gtk3

Sintaxis

- obj <- <widgetName> Nuevo: cómo se crean los widgets (por ejemplo, Windows, botones, cuadrículas)
- set <widget> [<atributos>] - establece atributos como se define como Attr self en la documentación del widget (por ejemplo, buttonLabel)
- en <widget> <evento> <acción IO> - Agregar una acción IO a un widget Señal self (por ejemplo, botón activado)

Observaciones

En muchas distribuciones de Linux, la biblioteca Haskell Gtk3 está disponible como un paquete en el administrador de paquetes del sistema (por ejemplo, `libghc-gtk` en el APT de Ubuntu). Sin embargo, para algunos desarrolladores que podría ser preferible utilizar una herramienta como `stack` para gestionar entornos aislados y tener instalado gtk3 a través de `cabal` en lugar de a través de una instalación global mediante el gestor de paquetes de sistemas. Para esta opción, se requiere `gtk2hs-buildtools` . Ejecute `cabal install gtk2hs-buildtools` antes de agregar `gtk` , `gtk3` o cualquier otra biblioteca de Haskell basada en Gtk a la entrada de `build-depends` proyectos en su archivo de cabal.

Examples

Hola mundo en gtk

Este ejemplo muestra cómo se puede crear un simple "Hello World" en Gtk3, configurando una ventana y botones de botones. El código de muestra también mostrará cómo establecer diferentes atributos y acciones en los widgets.

```
module Main (Main.main) where

import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window <- windowNew
  on window objectDestroy mainQuit
  set window [ containerBorderWidth := 10, windowTitle := "Hello World" ]
  button <- buttonNew
  set button [ buttonLabel := "Hello World" ]
  on button buttonActivated $ do
    putStrLn "A \"clicked\"-handler to say \"destroy\""
    widgetDestroy window
  set window [ containerChild := button ]
```

```
  widgetShowAll window
  mainGUI -- main loop
```

Lea Gtk3 en línea: https://riptutorial.com/es/haskell/topic/7342/gtk3

Capítulo 36: Interfaz de función extranjera

Sintaxis

- importación extranjera ccall inseguro "foo" hFoo :: Int32 -> IO Int32 {- Importa una función llamada `foo` en algún archivo de objeto, y define el símbolo `hFoo` que se puede llamar con el código de Haskell. -}

Observaciones

Si bien cabal tiene soporte para incluir bibliotecas de C y C ++ en un paquete de Haskell, hay algunos errores. Primero, si tiene datos (en lugar de una función) definidos en `bo` que se usan en `ao` , y liste las `C-sources: ac, bc` , entonces Cabal no podrá encontrar los datos. Esto está documentado en # 12152 . Una solución alternativa al utilizar cabal es reordenar la lista de `C-sources` `C-sources: bc, ac` para que sean `C-sources: bc, ac` . Es posible que esto no funcione cuando se usa la pila, porque la pila siempre vincula las `C-sources` alfabéticamente, independientemente del orden en el que se enumeran.

Otro problema es que debe rodear cualquier código C ++ en los archivos de encabezado (.h) con guardias `#ifdef __cplusplus` . Esto se debe a que GHC no entiende el código C ++ en los archivos de encabezado. Aún puede escribir código C ++ en archivos de encabezado, pero debe rodearlo con guardas.

`ccall` refiere a la *convención de llamada* ; actualmente se `ccall` y `stdcall` (convención de Pascal). La palabra clave `unsafe` es opcional; esto reduce la sobrecarga para funciones simples, pero puede causar interbloqueos si la función foránea se bloquea indefinidamente o no tiene suficiente permiso para ejecutar 1 .

Examples

Llamando C desde Haskell

Por razones de rendimiento, o debido a la existencia de bibliotecas C maduras, es posible que desee llamar al código C desde un programa de Haskell. Este es un ejemplo simple de cómo puede pasar datos a una biblioteca de C y obtener una respuesta.

foo.c:

```
#include <inttypes.h>

int32_t foo(int32_t a) {
  return a+1;
}
```

Foo.hs:

```
import Data.Int

main :: IO ()
main = print =<< hFoo 41

foreign import ccall unsafe "foo" hFoo :: Int32 -> IO Int32
```

La palabra clave `unsafe` genera una llamada más eficiente que "segura", pero requiere que el código C nunca realice una devolución de llamada al sistema Haskell. Ya que `foo` está completamente en C y nunca llamará a Haskell, podemos usar `unsafe` .

También necesitamos dar instrucciones a Cabal para compilar y enlazar en C fuente.

foo.cabal:

```
name:                foo
version:             0.0.0.1
build-type:          Simple
extra-source-files:  *.c
cabal-version:       >= 1.10

executable foo
  default-language: Haskell2010
  main-is:       Foo.hs
  C-sources:     foo.c
  build-depends: base
```

Entonces puedes correr:

```
> cabal configure
> cabal build foo
> ./dist/build/foo/foo
42
```

Pasar funciones de Haskell como devoluciones de llamada a código C.

Es muy común que las funciones C acepten punteros a otras funciones como argumentos. El ejemplo más popular es configurar una acción para que se ejecute cuando se hace clic en un botón en alguna biblioteca de herramientas de GUI. Es posible pasar las funciones de Haskell como C callbacks.

Para llamar a esta función C:

```
void event_callback_add (Object *obj, Object_Event_Cb func, const void *data)
```

Primero lo importamos al código de Haskell:

```
foreign import ccall "header.h event_callback_add"
    callbackAdd :: Ptr () -> FunPtr Callback -> Ptr () -> IO ()
```

Ahora, observando cómo se define `Object_Event_Cb` en el encabezado C, defina qué `Callback` está

en Haskell:

```
type Callback = Ptr () -> Ptr () -> IO ()
```

Finalmente, cree una función especial que envuelva la función Haskell de tipo `Callback` en un puntero `FunPtr Callback` :

```
foreign import ccall "wrapper"
    mkCallback :: Callback -> IO (FunPtr Callback)
```

Ahora podemos registrar el callback con código C:

```
cbPtr <- mkCallback $ \objPtr dataPtr -> do
    -- callback code
    return ()
callbackAdd cpPtr
```

Es importante liberar `FunPtr` asignado una vez que `FunPtr` registro de la devolución de llamada:

```
freeHaskellFunPtr cbPtr
```

Lea Interfaz de función extranjera en línea: https://riptutorial.com/es/haskell/topic/7256/interfaz-de-funcion-extranjera

Capítulo 37: IO

Examples

Leyendo todos los contenidos de entrada estándar en una cadena

```
main = do
    input <- getContents
    putStr input
```

Entrada:

```
This is an example sentence.
And this one is, too!
```

Salida:

```
This is an example sentence.
And this one is, too!
```

Nota: este programa realmente imprimirá partes de la salida antes de que se haya leído completamente toda la entrada. Esto significa que, si, por ejemplo, usa `getContents` sobre un archivo de 50MiB, la evaluación perezosa de Haskell y el recolector de basura se asegurarán de que solo el Las partes del archivo que se necesitan actualmente (lectura: indispensable para una ejecución posterior) se cargarán en la memoria. Por lo tanto, el archivo 50MiB no se cargará en la memoria de una vez.

Leyendo una línea de entrada estándar

```
main = do
    line <- getLine
    putStrLn line
```

Entrada:

```
This is an example.
```

Salida:

```
This is an example.
```

Analizar y construir un objeto desde la entrada estándar

```
readFloat :: IO Float
readFloat =
    fmap read getLine
```

```
main :: IO ()
main = do
    putStr "Type the first number: "
    first <- readFloat

    putStr "Type the second number: "
    second <- readFloat

    putStrLn $ show first ++ " + " ++ show second ++ " = " ++ show ( first + second )
```

Entrada:

```
Type the first number: 9.5
Type the second number: -2.02
```

Salida:

```
9.5 + -2.02 = 7.48
```

Leyendo desde los manejadores de archivos

Al igual que en otras partes de la biblioteca de E / S, las funciones que utilizan implícitamente un flujo estándar tienen una contraparte en `System.IO` que realiza el mismo trabajo, pero con un parámetro adicional a la izquierda, de tipo `Handle` , que representa el flujo que se está manejado. Por ejemplo, `getLine :: IO String` tiene una contraparte `hGetLine :: Handle -> IO String` .

```
import System.IO( Handle, FilePath, IOMode( ReadMode ),
                  openFile, hGetLine, hPutStr, hClose, hIsEOF, stderr )

import Control.Monad( when )


dumpFile :: Handle -> FilePath -> Integer -> IO ()

dumpFile handle filename lineNumber = do      -- show file contents line by line
    end <- hIsEOF handle
    when ( not end ) $ do
        line <- hGetLine handle
        putStrLn $ filename ++ ":" ++ show lineNumber ++ ": " ++ line
        dumpFile handle filename $ lineNumber + 1


main :: IO ()

main = do
    hPutStr stderr "Type a filename: "
    filename <- getLine
    handle <- openFile filename ReadMode
    dumpFile handle filename 1
    hClose handle
```

Contenido del archivo `example.txt` :

```
This is an example.
Hello, world!
This is another example.
```

Entrada:

```
Type a filename: example.txt
```

Salida:

```
example.txt:1: This is an example.
example.txt:2: Hello, world!
example.txt:3: This is another example
```

Comprobación de condiciones de fin de archivo

Un poco contrario a la intuición de la forma en que lo hacen las bibliotecas de E / S estándar de otros idiomas, el `isEOF` de Haskell no requiere que realice una operación de lectura antes de verificar una condición de EOF; El tiempo de ejecución lo hará por ti.

```
import System.IO( isEOF )


eofTest :: Int -> IO ()
eofTest line = do
    end <- isEOF
    if end then
        putStrLn $ "End-of-file reached at line " ++ show line ++ "."
    else do
        getLine
        eofTest $ line + 1


main :: IO ()
main =
    eofTest 1
```

Entrada:

```
Line #1.
Line #2.
Line #3.
```

Salida:

```
End-of-file reached at line 4.
```

Leyendo palabras de un archivo completo

En Haskell, a menudo tiene sentido *no molestarse en manejar los archivos* , sino simplemente leer o escribir un archivo completo directamente desde el disco a la memoria † , y hacer toda la

partición / procesamiento del texto con la estructura de datos de cadena pura. Esto evita mezclar IO y la lógica del programa, lo que puede ayudar enormemente a evitar errores.

```
-- | The interesting part of the program, which actually processes data
--   but doesn't do any IO!
reverseWords :: String -> [String]
reverseWords = reverse . words

-- | A simple wrapper that only fetches the data from disk, lets
--   'reverseWords' do its job, and puts the result to stdout.
main :: IO ()
main = do
   content <- readFile "loremipsum.txt"
   mapM_ putStrLn $ reverseWords content
```

Si `loremipsum.txt` contiene

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit
```

entonces el programa dará salida

```
elit
adipiscing
consectetur
amet,
sit
dolor
ipsum
Lorem
```

Aquí, `mapM_` la lista de todas las palabras en el archivo e imprimió cada una de ellas en una línea separada con `putStrLn` .

---

† Si crees que esto es un desperdicio de memoria, tienes un punto. En realidad, la pereza de Haskell a menudo puede evitar que todo el archivo deba residir en la memoria simultáneamente ... pero tenga cuidado, este tipo de IO perezosa causa su propio conjunto de problemas. Para aplicaciones de rendimiento crítico, a menudo tiene sentido imponer el archivo completo para que se lea a la vez, estrictamente; se puede hacer esto con el `Data.Text` versión de `readFile` .

IO define la acción `main` de su programa.

Para hacer un programa ejecutable de Haskell, debe proporcionar un archivo con una función `main` de tipo `IO ()`

```
main :: IO ()
main = putStrLn "Hello world!"
```

Cuando Haskell se compila, examina los datos de `IO` aquí y los convierte en un ejecutable. Cuando ejecutemos este programa se imprimirá `Hello world!` .

Si tiene valores de tipo `IO a` no sean `main` , no harán nada.

```
other :: IO ()
other = putStrLn "I won't get printed"

main :: IO ()
main = putStrLn "Hello world!"
```

Compilar este programa y ejecutarlo tendrá el mismo efecto que el último ejemplo. El código en `other` se ignora.

Para hacer que el código en `other` tenga efectos de tiempo de ejecución, debe *componerlo* en `main` . Ningún valor `IO` que no se componga finalmente en `main` tendrá ningún efecto de tiempo de ejecución. Para componer dos valores de `IO` secuencial, puede usar `do` notation:

```
other :: IO ()
other = putStrLn "I will get printed... but only at the point where I'm composed into main"

main :: IO ()
main = do
  putStrLn "Hello world!"
  other
```

Cuando compilas y ejecutas *este* programa sale

```
Hello world!
I will get printed... but only at the point where I'm composed into main
```

Tenga en cuenta que el orden de las operaciones se describe según la composición de los `other` en `main` y no en el orden de definición.

Papel y propósito de IO

Haskell es un lenguaje puro, lo que significa que las expresiones no pueden tener efectos secundarios. Un efecto secundario es cualquier cosa que la expresión o función haga que no sea producir un valor, por ejemplo, modificar un contador global o imprimir en una salida estándar.

En Haskell, los cálculos de efectos secundarios (específicamente, aquellos que pueden tener un efecto en el mundo real) se modelan utilizando `IO` . En sentido estricto, `IO` es un constructor de tipos, que toma un tipo y produce un tipo. Por ejemplo, `IO Int` es el tipo de cálculo de E / S que produce un valor `Int` . El tipo de `IO` es *abstracto* y la interfaz provista para `IO` garantiza que no puedan existir ciertos valores ilegales (es decir, funciones con tipos no sensitivos), al garantizar que todas las funciones incorporadas que realizan IO tienen un tipo de retorno incluido en `IO` .

Cuando se ejecuta un programa de Haskell, se ejecuta el cálculo representado por el valor de Haskell denominado `main` , cuyo tipo puede ser `IO x` para cualquier tipo `x` .

Manipulando valores IO

Hay muchas funciones en la biblioteca estándar que proporcionan acciones típicas de `IO` que debe realizar un lenguaje de programación de propósito general, como leer y escribir en los manejadores de archivos. Las acciones generales de `IO` se crean y combinan principalmente con dos funciones:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Esta función (normalmente llamada *vinculación*) toma una acción de `IO` y una función que devuelve una acción de `IO` , y produce la acción de `IO` que es el resultado de aplicar la función al valor producido por la primera acción de `IO` .

```
return :: a -> IO a
```

Esta función toma cualquier valor (es decir, un valor puro) y devuelve el cálculo de IO que no hace IO y produce el valor dado. En otras palabras, es una acción de E / S sin operación.

Hay funciones generales adicionales que se usan a menudo, pero todas pueden escribirse en los términos de los dos anteriores. Por ejemplo, `(>>) :: IO a -> IO b -> IO b` es similar a `(>>=)` pero se ignora el resultado de la primera acción.

Un programa simple que saluda al usuario usando estas funciones:

```
main :: IO ()
main =
  putStrLn "What is your name?" >>
  getLine >>= \name ->
  putStrLn ("Hello " ++ name ++ "!")
```

Este programa también usa `putStrLn :: String -> IO ()` y `getLine :: IO String` .

Nota: los tipos de ciertas funciones anteriores son en realidad más generales que los tipos dados (a saber, `>>=` , `>>` y `return`).

Semántica IO

El tipo `IO` en Haskell tiene una semántica muy similar a la de los lenguajes de programación imperativos. Por ejemplo, cuando uno escribe `s1 ; s2` en un lenguaje imperativo para indicar la ejecución de la instrucción `s1` , luego la instrucción `s2` , se puede escribir `s1 >> s2` para modelar lo mismo en Haskell.

Sin embargo, la semántica de `IO` divergente ligeramente de lo que se esperaría que provenga de un fondo imperativo. La función de `return` *no* interrumpe el flujo de control; no tiene ningún efecto en el programa si se ejecuta otra acción de `IO` en secuencia. Por ejemplo, `return () >> putStrLn "boom"` imprime correctamente "boom" en la salida estándar.

La semántica formal de `IO` puede dar en términos de ecualizaciones simples que involucran las

funciones en la sección anterior:

```
return x >>= f ≡ f x, ∀ f x
y >>= return ≡ return y, ∀ y
(m >>= f) >>= g ≡ m >>= (\x -> (f x >>= g)), ∀ m f g
```

Estas leyes suelen denominarse identidad de izquierda, identidad de derecha y composición, respectivamente. Se pueden afirmar más naturalmente en términos de la función.

```
(>=>) :: (a -> IO b) -> (b -> IO c) -> a -> IO c
(f >=> g) x = (f x) >>= g
```

como sigue:

```
return >=> f ≡ f, ∀ f
f >=> return ≡ f, ∀ f
(f >=> g) >=> h ≡ f >=> (g >=> h), ∀ f g h
```

Perezoso IO

Las funciones que realizan cálculos de E / S suelen ser estrictas, lo que significa que todas las acciones anteriores en una secuencia de acciones deben completarse antes de que comience la siguiente acción. Normalmente, este es un comportamiento útil y esperado - `putStrLn "X" >> putStrLn "Y"` debe imprimir "XY". Sin embargo, ciertas funciones de la biblioteca realizan la E / S perezosamente, lo que significa que las acciones de E / S requeridas para producir el valor solo se realizan cuando el valor se consume realmente. Ejemplos de tales funciones son `getContents` y `readFile` . La perezosa de E / S puede reducir drásticamente el rendimiento de un programa Haskell, por lo que al usar las funciones de la biblioteca, se debe tener cuidado de observar qué funciones son perezosas.

IO y `do` la notación

Haskell proporciona un método más simple para combinar diferentes valores de IO en valores de IO más grandes. Esta sintaxis especial se conoce como `do` la notación * y es simplemente azúcar sintáctica para los usos de la `>>=` , `>>` y `return` funciones.

El programa en la sección anterior se puede escribir de dos maneras diferentes usando la notación `do` , la primera es sensible al diseño y la segunda no sensible al diseño:

```
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello " ++ name ++ "!")


main = do {
```

```
  putStrLn "What is your name?" ;
  name <- getLine ;
  putStrLn ("Hello " ++ name ++ "!")
  }
```

Los tres programas son exactamente equivalentes.

---

* Tenga en cuenta que la notación `do` también es aplicable a una clase más amplia de constructores de tipo llamados *mónadas* .

Obtener la 'a' "de" 'IO a'

Una pregunta común es "tengo un valor de `IO a` , pero quiero hacer algo para que `a` valor:? ¿Cómo puedo obtener acceso a ella" ¿Cómo se puede operar con datos que provienen del mundo exterior (por ejemplo, incrementando un número escrito por el usuario)?

El punto es que si utiliza una función pura en los datos obtenidos de forma impura, el resultado sigue siendo impuro. ¡Depende de lo que hizo el usuario! Un valor de tipo `IO a` significa un "cálculo de efectos secundarios que resulta en un valor de tipo `a` " que *solo* puede ejecutarse (a) componiéndolo en `main` y (b) compilando y ejecutando su programa. Por esa razón, no hay manera dentro del mundo Haskell puro para "conseguir el `a` cabo".

En su lugar, queremos construir un nuevo cálculo, un nuevo `IO` valor, que hace uso de la `a` valor *en tiempo de ejecución.* Esta es otra forma de *componer* valores de `IO` y, de nuevo, podemos usar `do` notation:

```
-- assuming
myComputation :: IO Int

getMessage :: Int -> String
getMessage int = "My computation resulted in: " ++ show int

newComputation :: IO ()
newComputation = do
  int <- myComputation       -- we "bind" the result of myComputation to a name, 'int'
  putStrLn $ getMessage int   -- 'int' holds a value of type Int
```

Aquí estamos usando una función pura (`getMessage`) para convertir un `Int` en un `String` , pero estamos usando la notación `do` para hacer que se aplique al resultado de un cálculo de `IO` `myComputation` *cuando* (después) se ejecuta ese cálculo. El resultado es un cálculo de `IO` más grande, `newComputation` . Esta técnica de usar funciones puras en un contexto impuro se llama *levantamiento* .

Escribiendo al stdout

Según la especificación de idioma de Haskell 2010, las siguientes son funciones de IO estándar disponibles en Prelude, por lo que no se requieren importaciones para usarlas.

`putChar :: Char -> IO ()` - escribe un `char` en `stdout`

```
Prelude> putChar 'a'
aPrelude>  -- Note, no new line
```

`putStr :: String -> IO ()` - escribe un `String` en `stdout`

```
Prelude> putStr "This is a string!"
This is a string!Prelude>  -- Note, no new line
```

`putStrLn :: String -> IO ()` - escribe un `String` en `stdout` y agrega una nueva línea

```
Prelude> putStrLn "Hi there, this is another String!"
Hi there, this is another String!
```

`print :: Show a => a -> IO ()` - escribe `a` instancia de `Show` a `stdout`

```
Prelude> print "hi"
"hi"
Prelude> print 1
1
Prelude> print 'a'
'a'
Prelude> print (Just 'a')  -- Maybe is an instance of the `Show` type class
Just 'a'
Prelude> print Nothing
Nothing
```

Recuerde que puede crear una instancia de `Show` para sus propios tipos utilizando la `deriving` :

```
-- In ex.hs
data Person = Person { name :: String } deriving Show
main = print (Person "Alex")  -- Person is an instance of `Show`, thanks to `deriving`
```

Cargando y corriendo en GHCi:

```
Prelude> :load ex.hs
[1 of 1] Compiling ex             ( ex.hs, interpreted )
Ok, modules loaded: ex.
*Main> main  -- from ex.hs
Person {name = "Alex"}
*Main>
```

Leyendo de `stdin`

Según la especificación de lenguaje de Haskell 2010 , las siguientes son funciones de IO estándar disponibles en Prelude, por lo que no se requieren importaciones para usarlas.

`getChar :: IO Char` - lee un `Char` de `stdin`

```
-- MyChar.hs
```

```
main = do
  myChar <- getChar
  print myChar

-- In your shell

runhaskell MyChar.hs
a -- you enter a and press enter
'a'  -- the program prints 'a'
```

`getLine :: IO String` - lee un `String` desde `stdin` , sin una nueva línea de caracteres

```
Prelude> getLine
Hello there!  -- user enters some text and presses enter
"Hello there!"
```

`read :: Read a => String -> a` - convierte un String a un valor

```
Prelude> read "1" :: Int
1
Prelude> read "1" :: Float
1.0
Prelude> read "True" :: Bool
True
```

Otras funciones menos comunes son:

- `getContents :: IO String` - devuelve todas las entradas del usuario como una sola cadena, que se lee con pereza a medida que se necesita
- `interact :: (String -> String) -> IO ()` - toma una función de tipo String -> String como su argumento. La entrada completa del dispositivo de entrada estándar se pasa a esta función como su argumento

Lea IO en línea: https://riptutorial.com/es/haskell/topic/1904/io

Capítulo 38: Lector / Lector

Introducción

El lector proporciona funcionalidad para pasar un valor a lo largo de cada función. Una guía útil con algunos diagramas puede encontrarse aquí: http://adit.io/posts/2013-06-10-three-useful-monads.html

Examples

Simple demostración

Una parte clave de la mónada de Reader es la función `ask` (https://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Reader.html#v:ask) , que se define con fines ilustrativos. fines:

```
import Control.Monad.Trans.Reader hiding (ask)
import Control.Monad.Trans

ask :: Monad m => ReaderT r m r
ask = reader id

main :: IO ()
main = do
  let f = (runReaderT $ readerExample) :: Integer -> IO String
  x <- f 100
  print x
  --
  let fIO = (runReaderT $ readerExampleIO) :: Integer -> IO String
  y <- fIO 200
  print y

readerExample :: ReaderT Integer IO String
readerExample = do
  x <- ask
  return $ "The value is: " ++ show x

liftAnnotated :: IO a -> ReaderT Integer IO a
liftAnnotated = lift

readerExampleIO :: ReaderT Integer IO String
readerExampleIO = do
  x <- reader id
  lift $ print "Hello from within"
  liftAnnotated $ print "Hello from within..."
  return $ "The value is: " ++ show x
```

Lo anterior se imprimirá:

```
"The value is: 100"
"Hello from within"
"Hello from within..."
```

```
"The value is: 200"
```

Lea Lector / Lector en línea: https://riptutorial.com/es/haskell/topic/9320/lector---lector

Capítulo 39: Lente

Introducción

Lens es una biblioteca para Haskell que proporciona lentes, isomorfismos, pliegues, recorridos, captadores y definidores, lo que expone una interfaz uniforme para consultar y manipular estructuras arbitrarias, no como los conceptos de acceso y mutación de Java.

Observaciones

¿Qué es una lente?

Las lentes (y otras ópticas) nos permiten separarnos describiendo *cómo* queremos acceder a algunos datos de *lo* que queremos hacer con ellos. Es importante distinguir entre la noción abstracta de una lente y la implementación concreta. La comprensión abstracta hace que la programación con `lens` sea mucho más fácil a largo plazo. Hay muchas representaciones isomorfas de lentes, por lo que para esta discusión evitaremos cualquier discusión de implementación concreta y, en cambio, ofreceremos una visión general de alto nivel de los conceptos.

Enfoque

Un concepto importante en la comprensión abstracta es la noción de *enfoque* . Las ópticas importantes se *centran* en una parte específica de una estructura de datos más grande sin olvidar el contexto más amplio. Por ejemplo, la lente `_1` enfoca en el primer elemento de una tupla pero no se olvida de lo que había en el segundo campo.

Una vez que tenemos el enfoque, podemos hablar sobre qué operaciones se nos permite realizar con una lente. Dado un `Lens sa` que cuando se da un tipo de datos de tipo `s` se centra en un determinado `a` , podemos ya sea

1. Extraiga la `a` olvidando el contexto adicional o
2. Reemplace el `a` proporcionando un nuevo valor

Estos corresponden a las conocidas operaciones de `get` y `set` que normalmente se utilizan para caracterizar una lente.

Otras ópticas

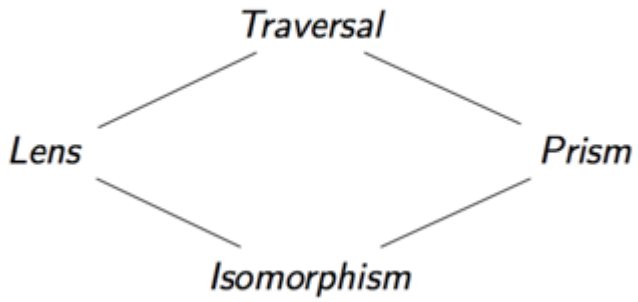
Podemos hablar de otras ópticas de manera similar.

Óptico	Se centra en...
Lente	Una parte de un producto
Prisma	Una parte de una suma
Travesía	Cero o más partes de una estructura de datos
Isomorfismo	...

Cada óptica se enfoca de una manera diferente, como tal, dependiendo de qué tipo de óptica tengamos podemos realizar diferentes operaciones.

Composición

Además, podemos componer cualquiera de las dos ópticas que hemos discutido hasta ahora para especificar accesos de datos complejos. Los cuatro tipos de ópticas que hemos discutido forman una red, el resultado de componer dos ópticas juntas es su límite superior.



Por ejemplo, si componemos juntos una lente y un prisma, obtenemos un recorrido transversal. La razón de esto es que por su composición (vertical), primero nos enfocamos en una parte de un producto y luego en una parte de una suma. El resultado es una óptica que se centra precisamente en cero o en una parte de nuestros datos, que es un caso especial de un recorrido. (Esto a veces también se llama un recorrido afín).

En haskell

La razón de la popularidad en Haskell es que hay una representación muy breve de la óptica. Todas las ópticas son solo funciones de una cierta forma que se pueden componer juntas utilizando la función de composición. Esto lleva a una incrustación muy ligera que facilita la integración de la óptica en sus programas. Además de esto, debido a los detalles de la codificación, la composición de la función también calcula automáticamente el límite superior de las dos ópticas que componemos. Esto significa que podemos reutilizar los mismos combinadores para diferentes ópticas sin una conversión explícita.

Examples

Manipulación de tuplas con lente

Consiguiendo

```
("a", 1) ^. _1 -- returns "a"
("a", 1) ^. _2 -- returns 1
```

Ajuste

```
("a", 1) & _1 .~ "b" -- returns ("b", 1)
```

Modificando

```
("a", 1) & _2 %~ (+1) -- returns ("a", 2)
```

`both` transversal

```
(1, 2) & both *~ 2 -- returns (2, 4)
```

Lentes para discos

Registro simple

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens

data Point = Point {
    _x :: Float,
    _y :: Float
}
makeLenses ''Point
```

Lentes `x` e `y` son creados.

```
let p = Point 5.0 6.0
p ^. x     -- returns 5.0
set x 10 p -- returns Point { _x = 10.0, _y = 6.0 }
p & x +~ 1 -- returns Point { _x = 6.0, _y = 6.0 }
```

Gestión de registros con nombres de campos repetidos

```
data Person = Person { _personName :: String }
makeFields ''Person
```

Crea una clase de tipo `HasName` , `name` lente para `Person` , y convierte a `Person` una instancia de `HasName` . Los registros subsiguientes también se agregarán a la clase:

```
data Entity = Entity { _entityName :: String }
makeFields ''Entity
```

Se requiere la extensión Template Haskell para que `makeFields` funcione. Técnicamente, es totalmente posible crear las lentes hechas de esta manera a través de otros medios, por ejemplo, a mano.

Lentes de estado

Los operadores de lentes tienen variantes útiles que operan en contextos con estado. Se obtienen reemplazando `~` con `=` en el nombre del operador.

```
(+~) :: Num a => ASetter s t a a -> a -> s -> t
(+=) :: (MonadState s m, Num a) => ASetter' s a -> a -> m ()
```

> Nota: No se espera que las variantes con estado cambien el tipo, por lo que tienen las firmas de la `Lens'` o la `Simple Lens'` .

Deshacerse de `&` cadenas

Si es necesario encadenar las operaciones con lentes, a menudo se ve así:

```
change :: A -> A
change a = a & lensA %~ operationA
             & lensB %~ operationB
             & lensC %~ operationC
```

Esto funciona gracias a la asociatividad de `&` . Sin embargo, la versión con estado es más clara.

```
change a = flip execState a $ do
    lensA %= operationA
    lensB %= operationB
    lensC %= operationC
```

Si `lensX` es realmente `id` , toda la operación puede, por supuesto, ejecutarse directamente simplemente levantándola con `modify` .

Código imperativo con estado estructurado.

Asumiendo este estado de ejemplo:

```
data Point = Point { _x :: Float, _y :: Float }
data Entity = Entity { _position :: Point, _direction :: Float }
data World = World { _entities :: [Entity] }

makeLenses ''Point
makeLenses ''Entity
makeLenses ''World
```

Podemos escribir código que se asemeja a los lenguajes imperativos clásicos, mientras nos permite usar los beneficios de Haskell:

```
updateWorld :: MonadState World m => m ()
updateWorld = do
    -- move the first entity
    entities . ix 0 . position . x += 1

    -- do some operation on all of them
    entities . traversed . position %= \p -> p `pointAdd` ...

    -- or only on a subset
    entities . traversed . filtered (\e -> e ^. position.x > 100) %= ...
```

Escribiendo una lente sin plantilla Haskell

Para desmitificar Template Haskell, supongamos que tienes

```
data Example a = Example { _foo :: Int, _bar :: a }
```

entonces

```
makeLenses 'Example
```

produce (más o menos)

```
foo :: Lens' (Example a) Int
bar :: Lens (Example a) (Example b) a b
```

Sin embargo, no hay nada particularmente mágico en marcha. Puedes escribirlas tú mismo:

```
foo :: Lens' (Example a) Int
--  :: Functor f => (Int -> f Int) -> (Example a -> f (Example a))    ;; expand the alias
foo wrap (Example foo bar) = fmap (\newFoo -> Example newFoo bar) (wrap foo)

bar :: Lens (Example a) (Example b) a b
--  :: Functor f => (a -> f b) -> (Example a -> f (Example b))    ;; expand the alias
bar wrap (Example foo bar) = fmap (\newBar -> Example foo newBar) (wrap bar)
```

Esencialmente, usted quiere "visitar" el "enfoque" de su lente con la función de `wrap` y luego reconstruir el tipo "completo".

Lente y prisma

Una `Lens' sa` significa que *siempre* puedes encontrar una `a` dentro de cualquier `s` . Un `Prism' sa` significa que a *veces se* puede encontrar que `s` en realidad sólo *es* `a` pero a veces es otra cosa.

Para ser más claros, tenemos `_1 :: Lens' (a, b) a` porque cualquier tupla *siempre* tiene un primer elemento. Tenemos `_Just :: Prism' (Maybe a) a` porque *a veces* `Maybe a` es en realidad una `a` valor envuelto en `Just` pero *a veces* es `Nothing` .

Con esta intuición, algunos combinadores estándar se pueden interpretar paralelos entre sí.

- `view :: Lens' sa -> (s -> a)` "obtiene" la `a` fuera de la `s`
- `set :: Lens' sa -> (a -> s -> s)` "conjuntos" de la `a` ranura en `s`
- `review :: Prism' sa -> (a -> s)` "se da cuenta" de que una `a` podría ser una `s`
- `preview :: Prism' sa -> (s -> Maybe a)` "intenta" convertir un `s` en un `a` .

Otra forma de pensarlo es que un valor de tipo `Lens' sa` demuestra que `s` tiene la misma estructura que `(r, a)` para algunos `r` desconocidos. Por otro lado, `Prism' sa s` demuestra que `s` tiene la misma estructura que `Either ra` para alguna `r` . Podemos escribir las cuatro funciones anteriores con este conocimiento:

```
-- `Lens' s a` is no longer supplied, instead we just *know* that `s ~ (r, a)`

view :: (r, a) -> a
view (r, a) = a

set :: a -> (r, a) -> (r, a)
set a (r, _) = (r, a)

-- `Prism' s a` is no longer supplied, instead we just *know* that `s ~ Either r a`

review :: a -> Either r a
review a = Right a

preview :: Either r a -> Maybe a
preview (Left _) = Nothing
preview (Right a) = Just a
```

Travesías

Un `Traversal' sa` muestra que `s` tiene 0-a-muchos `a` s dentro de ella.

```
toListOf :: Traversal' s a -> (s -> [a])
```

Cualquier tipo `t` que sea `Traversable` tiene automáticamente ese `traverse :: Traversal (ta) a` .

Podemos utilizar un `Traversal` para establecer o mapa sobre todos ellos `a` valores

```
> set traverse 1 [1..10]
[1,1,1,1,1,1,1,1,1,1]

> over traverse (+1) [1..10]
[2,3,4,5,6,7,8,9,10,11]
```

A `f :: Lens' sa` dice que hay exactamente una `a` dentro de `s` . A `g :: Prism' ab` dice que hay 0 o 1 `b` s en `a` . Componiendo `f . g` nos da un `Traversal' sb` porque a continuación `f g` muestra cómo hay 0 a 1 `b` s en `s` .

Lentes componen

Si tienes un `f :: Lens' ab` y un `g :: Lens' bc` entonces `f . g` es `Lens' ac` una `Lens' ac` al seguir `f` primero y luego `g` . Notablemente:

- Lentes de componer como funciones (en realidad sólo *son* funciones)
- Si piensa en la funcionalidad de `view` de la `Lens` , parece que los datos fluyen "de izquierda a derecha": esto puede parecerle a su intuición normal para la composición de la función. Por otro lado, debería sentirse natural si piensas en ello . -Nota como ocurre en los idiomas OO.

Más que solo componer `Lens` with `Lens` , `(.)` Se puede usar para componer casi cualquier tipo " `Lens` like". No siempre es fácil ver cuál es el resultado, ya que el tipo se vuelve más difícil de seguir, pero puede usar la tabla de `lens` para averiguarlo. La composición `x . y` tiene el tipo del límite mínimo superior de los tipos tanto de `x` como de `y` en ese gráfico.

Lentes con clase

Además de la función estándar de `makeLenses` para generar `Lens` , `Control.Lens.TH` también ofrece la función `makeClassy` . `makeClassy` tiene el mismo tipo y funciona esencialmente de la misma manera que `makeLenses` , con una diferencia clave. Además de generar las lentes y recorridos estándar, si el tipo no tiene argumentos, también creará una clase que describa todos los tipos de datos que poseen el tipo como campo. Por ejemplo

```
data Foo = Foo { _fooX, _fooY :: Int }
  makeClassy ''Foo
```

creará

```
class HasFoo t where
   foo :: Simple Lens t Foo

instance HasFoo Foo where foo = id

fooX, fooY :: HasFoo t => Simple Lens t Int
```

Campos con makeFields

(Este ejemplo copiado de esta respuesta StackOverflow)

Digamos que tiene varios tipos de datos diferentes que todos deberían tener una lente con el mismo nombre, en este caso la `capacity` . La `makeFields` creará una clase que logrará esto sin conflictos de espacio de nombres.

```
{-# LANGUAGE FunctionalDependencies
           , MultiParamTypeClasses
```

```
           , TemplateHaskell
  #-}

module Foo
where

import Control.Lens

data Foo
  = Foo { fooCapacity :: Int }
  deriving (Eq, Show)
$(makeFields ''Foo)

data Bar
  = Bar { barCapacity :: Double }
  deriving (Eq, Show)
$(makeFields ''Bar)
```

Luego en ghci:

```
*Foo
λ let f = Foo 3
|     b = Bar 7
|
b :: Bar
f :: Foo

*Foo
λ fooCapacity f
3
it :: Int

*Foo
λ barCapacity b
7.0
it :: Double

*Foo
λ f ^. capacity
3
it :: Int

*Foo
λ b ^. capacity
7.0
it :: Double

λ :info HasCapacity
class HasCapacity s a | s -> a where
  capacity :: Lens' s a
    -- Defined at Foo.hs:14:3
instance HasCapacity Foo Int -- Defined at Foo.hs:14:3
instance HasCapacity Bar Double -- Defined at Foo.hs:19:3
```

Entonces, lo que realmente se hace se declara una clase `HasCapacity sa` , donde la capacidad es un `Lens'` de `s` a `a` (`a` se fija una vez que se conoce s). Descubrió el nombre "capacidad" al eliminar el nombre (en minúsculas) del tipo de datos del campo; Me resulta agradable no tener que usar un guión bajo ni en el nombre del campo ni en el de la lente, ya que a veces la sintaxis de

grabación es lo que usted desea. Puede usar makeFieldsWith y las diferentes Reglas de la lente para tener algunas opciones diferentes para calcular los nombres de las lentes.

En caso de que ayude, use ghci -ddump-splices Foo.hs:

```
[1 of 1] Compiling Foo              ( Foo.hs, interpreted )
Foo.hs:14:3-18: Splicing declarations
    makeFields ''Foo
  ======>
    class HasCapacity s a | s -> a where
      capacity :: Lens' s a
    instance HasCapacity Foo Int where
      {-# INLINE capacity #-}
      capacity = iso (\ (Foo x_a7fG) -> x_a7fG) Foo
Foo.hs:19:3-18: Splicing declarations
    makeFields ''Bar
  ======>
    instance HasCapacity Bar Double where
      {-# INLINE capacity #-}
      capacity = iso (\ (Bar x_a7ne) -> x_a7ne) Bar
Ok, modules loaded: Foo.
```

Así que el primer empalme hizo la clase `HasCapcity` y agregó una instancia para Foo; el segundo usó la clase existente e hizo una instancia para Bar.

Esto también funciona si importa la clase `HasCapcity` desde otro módulo; `makeFields` puede agregar más instancias a la clase existente y distribuir sus tipos en múltiples módulos. Pero si lo usa de nuevo en otro módulo donde no haya importado la clase, creará una nueva clase (con el mismo nombre) y tendrá dos lentes de capacidad sobrecargadas que no son compatibles.

Lea Lente en línea: https://riptutorial.com/es/haskell/topic/891/lente

Capítulo 40: Lista de Comprensiones

Examples

Lista de comprensión básica

Haskell tiene listas de comprensión , que se parecen mucho a las comprensiones de conjunto en matemáticas e implementaciones similares en lenguajes imperativos como Python y JavaScript. En su forma más básica, las listas de comprensión toman la siguiente forma.

```
[ x | x <- someList ]
```

Por ejemplo

```
[ x | x <- [1..4] ]    -- [1,2,3,4]
```

Las funciones también se pueden aplicar directamente a x:

```
[ f x | x <- someList ]
```

Esto es equivalente a:

```
map f someList
```

Ejemplo:

```
[ x+1 | x <- [1..4]]    -- [2,3,4,5]
```

Patrones en Expresiones de Generador

Sin embargo, `x` en la expresión del generador no es solo variable, sino que puede ser cualquier patrón. En casos de desajuste de patrón, el elemento generado se omite y el procesamiento de la lista continúa con el siguiente elemento, actuando así como un filtro:

```
[x | Just x <- [Just 1, Nothing, Just 3]]     -- [1, 3]
```

Un generador con una variable `x` en su patrón crea un nuevo alcance que contiene todas las expresiones a su derecha, donde `x` se define como el elemento generado.

Esto significa que los guardias pueden codificarse como

```
[ x | x <- [1..4], even x] ==
[ x | x <- [1..4], () <- [() | even x]] ==
[ x | x <- [1..4], () <- if even x then [()] else []]
```

Guardias

Otra característica de las listas de comprensión son los guardias, que también actúan como filtros. Los guardias son expresiones booleanas y aparecen en el lado derecho de la barra en una lista de comprensión.

Su uso más básico es

```
[x    | p x]   ===   if p x then [x] else []
```

Cualquier variable utilizada en una guarda debe aparecer a su izquierda en la comprensión, o estar dentro del alcance. Asi que,

```
[ f x | x <- list, pred1 x y, pred2 x]     -- `y` must be defined in outer scope
```

que es equivalente a

```
map f (filter pred2 (filter (\x -> pred1 x y) list))          -- or,

-- ($ list) (filter (`pred1` y) >>> filter pred2 >>> map f)

-- list >>= (\x-> [x | pred1 x y]) >>= (\x-> [x | pred2 x]) >>= (\x -> [f x])
```

(el operador `>>=` es `infixl 1` , es decir, se asocia (está entre paréntesis) a la izquierda). Ejemplos:

```
[ x       | x <- [1..4], even x]           -- [2,4]

[ x^2 + 1 | x <- [1..100], even x ]        -- map (\x -> x^2 + 1) (filter even [1..100])
```

Generadores anidados

Las comprensiones de listas también pueden dibujar elementos de varias listas, en cuyo caso el resultado será la lista de todas las combinaciones posibles de los dos elementos, como si las dos listas se procesaran de forma *anidada* . Por ejemplo,

```
[ (a,b) | a <- [1,2,3], b <- ['a','b'] ]

-- [(1,'a'), (1,'b'), (2,'a'), (2,'b'), (3,'a'), (3,'b')]
```

Comprensiones paralelas

Con la extensión de lenguaje Parallel List Comprehensions ,

```
[(x,y) | x <- xs | y <- ys]
```

es equivalente a

```
zip xs ys
```

Ejemplo:

```
[(x,y) | x <- [1,2,3] | y <- [10,20]]

-- [(1,10),(2,20)]
```

Fijaciones locales

Las comprensiones de listas pueden introducir enlaces locales para las variables que contienen algunos valores provisionales:

```
[(x,y) | x <- [1..4], let y=x*x+1, even y]    -- [(1,2),(3,10)]
```

El mismo efecto se puede lograr con un truco,

```
[(x,y) | x <- [1..4], y <- [x*x+1], even y]   -- [(1,2),(3,10)]
```

Las comprensiones de `let` in list son recursivas, como siempre. Pero los enlaces del generador no lo son, lo que permite el *sombreado* :

```
[x | x <- [1..4], x <- [x*x+1], even x]       -- [2,10]
```

Hacer notación

Cualquier lista de comprensión puede ser codificado de manera correspondiente con la lista mónada de `do` la notación .

```
[f x | x <- xs]                f  <$> xs         do { x <- xs ; return (f x) }

[f x | f <- fs, x <- xs]       fs <*> xs         do { f <- fs ; x <- xs ; return (f x) }

[y   | x <- xs, y <- f x]      f  =<< xs         do { x <- xs ; y <- f x ; return y }
```

Los guardias se pueden manejar usando `Control.Monad.guard` :

```
[x   | x <- xs, even x]                          do { x <- xs ; guard (even x) ; return x }
```

Lea Lista de Comprensiones en línea: https://riptutorial.com/es/haskell/topic/4970/lista-de-comprensiones

Capítulo 41: Literales sobrecargados

Observaciones

Literales enteros

es un numeral **sin** un punto decimal

por ejemplo `0` , `1` , `42` , ...

se aplica implícitamente a `fromInteger` que forma parte de la clase de tipo `Num` , por lo que efectivamente tiene el tipo `Num a => a` , es decir, puede tener cualquier tipo que sea una instancia de `Num`

Literales fraccionales

es un numeral **con** un punto decimal

por ejemplo `0.0` , `-0.1111` , ...

se aplica implícitamente a `fromRational` que forma parte de la clase de tipo `Fractional` , por lo que de hecho tiene el tipo `a => a` , es decir, puede tener cualquier tipo que sea una instancia de `Fractional`

Literales de cuerda

Si agrega la extensión de lenguaje `OverloadedStrings` a *GHC* , puede tener la misma para `String` -literals que luego se aplican a `fromString` desde la clase de tipo `Data.String.IsString`

Esto se usa a menudo para reemplazar `String` con `Text` o `ByteString` .

Lista de literales

Las listas se pueden definir con la sintaxis literal `[1, 2, 3]` . En GHC 7.8 y más allá, esto también se puede usar para definir otras estructuras similares a listas con la extensión `OverloadedLists` .

Por defecto, el tipo de `[]` es:

```
> :t []
[] :: [t]
```

Con `OverloadedLists` , esto se convierte en:

```
[] :: GHC.Exts.IsList l => l
```

Examples

Numero entero

El tipo del literal.

```
Prelude> :t 1
1 :: Num a => a
```

Elegir un tipo concreto con anotaciones.

Puede especificar el tipo siempre que el tipo de destino sea `Num` con una *anotación* :

```
Prelude> 1 :: Int
1
it :: Int
Prelude> 1 :: Double
1.0
it :: Double
Prelude> 1 :: Word
1
it :: Word
```

si no el compilador se quejará

Preludio> 1 :: Cuerda

```
<interactive>:
    No instance for (Num String) arising from the literal `1'
    In the expression: 1 :: String
    In an equation for `it': it = 1 :: String
```

Numeral flotante

El tipo del literal.

```
Prelude> :t 1.0
1.0 :: Fractional a => a
```

Elegir un tipo concreto con anotaciones.

Puede especificar el tipo con una *anotación de tipo* . El único requisito es que el tipo debe tener una instancia `Fractional` .

```
Prelude> 1.0 :: Double
1.0
it :: Double
Prelude> 1.0 :: Data.Ratio.Ratio Int
1 % 1
it :: GHC.Real.Ratio Int
```

si no el compilador se quejará

```
Prelude> 1.0 :: Int
<interactive>:
    No instance for (Fractional Int) arising from the literal `1.0'
    In the expression: 1.0 :: Int
    In an equation for `it': it = 1.0 :: Int
```

Instrumentos de cuerda

El tipo del literal.

Sin ninguna extensión, el tipo de un literal de cadena, es decir, algo entre comillas dobles, es solo una cadena, también conocida como lista de caracteres:

```
Prelude> :t "foo"
"foo" :: [Char]
```

Sin embargo, cuando la extensión `OverloadedStrings` está habilitada, los literales de cadena se convierten en polimórficos, similar a los literales de números :

```
Prelude> :set -XOverloadedStrings
Prelude> :t "foo"
"foo" :: Data.String.IsString t => t
```

Esto nos permite definir valores de tipos similares a cadenas sin la necesidad de conversiones explícitas. En esencia, la extensión `OverloadedStrings` simplemente envuelve cada literal de cadena en la función de conversión genérica `fromString` , por lo que si el contexto exige, por ejemplo, el `Text` más eficiente en lugar de la `String` , no necesita preocuparse por eso.

Usando cadenas literales

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Text (Text, pack)
import Data.ByteString (ByteString, pack)


withString :: String
withString = "Hello String"

-- The following two examples are only allowed with OverloadedStrings

withText :: Text
```

```
withText = "Hello Text"      -- instead of: withText = Data.Text.pack "Hello Text"

withBS :: ByteString
withBS = "Hello ByteString"  -- instead of: withBS = Data.ByteString.pack "Hello ByteString"
```

Observe cómo pudimos construir valores de `Text` y `ByteString` de la misma manera en que construimos los valores de `String` (o `[Char]`) ordinarios, en lugar de usar cada función de `pack` tipos para codificar explícitamente la cadena.

Para obtener más información sobre la extensión de lenguaje `OverloadedStrings` , consulte la documentación de la extensión .

Lista de literales

La extensión OverloadedLists de GHC le permite construir estructuras de datos similares a listas con la sintaxis literal de listas.

Esto le permite a Data.Map como esto:

```
> :set -XOverloadedLists
> import qualified Data.Map as M
> M.lookup "foo" [("foo", 1), ("bar", 2)]
Just 1
```

En lugar de esto (note el uso de M.fromList extra):

```
> import Data.Map as M
> M.lookup "foo" (M.fromList [("foo", 1), ("bar", 2)])
Just 1
```

Lea Literales sobrecargados en línea: https://riptutorial.com/es/haskell/topic/369/literales-sobrecargados

Capítulo 42: Liza

Sintaxis

1. constructor de lista vacía

   ```
   [] :: [a]
   ```

2. constructor de listas no vacías

   ```
   (:) :: a -> [a] -> [a]
   ```

3. head - devuelve el primer valor de una lista

   ```
   head :: [a] -> a
   ```

4. last - devuelve el último valor de una lista

   ```
   last :: [a] -> a
   ```

5. cola - devuelve una lista sin el primer elemento

   ```
   tail :: [a] -> [a]
   ```

6. init - devuelve una lista sin el último elemento

   ```
   init :: [a] -> [a]
   ```

7. xs !! i - devuelve el elemento en un índice i en la lista xs

   ```
   (!!) :: Int -> [a] -> a
   ```

8. take n xs - devuelve una nueva lista que contiene n primeros elementos de la lista xs

   ```
   take :: Int -> [a] -> [a]
   ```

9. mapa :: (a -> b) -> [a] -> [b]
10. filtro :: (a -> Bool) -> [a] -> [a]
11. (++) :: [a] -> [a]
12. concat :: [[a]] -> [a]

Observaciones

1. El tipo `[a]` es equivalente a `[] a` .
2. `[]` construye la lista vacía.
3. `[]` en una definición de función LHS, por ejemplo, `f [] = ...` , es el patrón de lista vacía.
4. `x:xs` construye una lista donde un elemento `x` se añade a la lista `xs`
5. `f (x:xs) = ...` es una coincidencia de patrón para una lista no vacía donde `x` es la cabeza y `xs`

es la cola.
6. `f (a:b:cs) = ...` y `f (a:(b:cs)) = ...` son iguales. Son una coincidencia de patrón para una lista de al menos dos elementos donde el primer elemento es `a` , el segundo elemento es `b` , y el resto de la lista es `cs` .
7. `f ((a:as):bs) = ...` NO es lo mismo que `f (a:(as:bs)) = ...` La primera es una coincidencia de patrón para una lista de listas no vacías, donde `a` es la cabeza de la cabeza, `as` es la cola de la cabeza, y `bs` es la cola.
8. `f (x:[]) = ...` `f [x] = ...` son iguales. Son una coincidencia de patrón para una lista de exactamente un elemento.
9. `f (a:b:[]) = ...` y `f [a,b] = ...` son iguales. Son una coincidencia de patrón para una lista de exactamente dos elementos.
10. `f [a:b] = ...` es una coincidencia de patrón para una lista de exactamente un elemento donde el elemento también es una lista. `a` es la cabeza del elemento y `b` es la cola del elemento.
11. `[a,b,c]` es lo mismo que `(a:b:c:[])` . La notación de lista estándar es solo azúcar sintáctica para los constructores `(:)` y `[]` .
12. Puede usar `all@(x:y:ys)` para referirse a la lista completa como `all` (o cualquier otro nombre que elija) en lugar de repetir `(x:y:ys)` nuevamente.

Examples

Lista de literales

```
emptyList     = []

singletonList = [0]               -- = 0 : []

listOfNums    = [1, 2, 3]         -- = 1 : 2 : [3]

listOfStrings = ["A", "B", "C"]
```

Concatenación de listas

```
listA      = [1, 2, 3]

listB      = [4, 5, 6]

listAThenB = listA ++ listB       -- [1, 2, 3, 4, 5, 6]

(++) xs     [] = xs
(++) []     ys = ys
(++) (x:xs) ys = x : (xs ++ ys)
```

Conceptos básicos de la lista

El constructor de tipos para listas en el preludio de Haskell es `[]` . La declaración de tipo para una lista que contiene valores de tipo `Int` se escribe de la siguiente manera:

```
xs :: [Int]    -- or equivalently, but less conveniently,
xs :: [] Int
```

Las listas en Haskell son *secuencias homogéneas* , es decir, todos los elementos deben ser del mismo tipo. A diferencia de las tuplas, el tipo de lista no se ve afectado por la longitud:

```
[1,2,3]   :: [Int]
[1,2,3,4] :: [Int]
```

Las listas se construyen utilizando dos constructores :

- `[]` construye una lista vacía.
- `(:)` , pronunciado "contras", antepone elementos a una lista. Al considerar `x` (un valor de tipo `a`) sobre `xs` (una lista de valores del mismo tipo `a`) se crea una nueva lista, cuya *cabecera* (el primer elemento) es `x` , y la *cola* (el resto de los elementos) es `xs` .

Podemos definir listas simples de la siguiente manera:

```
ys :: [a]
ys = []

xs :: [Int]
xs = 12 : (99 : (37 : []))
-- or  = 12 : 99 : 37 : []     -- ((:) is right-associative)
-- or  = [12, 99, 37]          -- (syntactic sugar for lists)
```

Tenga en cuenta que `(++)` , que puede usarse para crear listas, se define recursivamente en términos de `(:)` y `[]` .

Procesando listas

Para procesar las listas, simplemente podemos hacer un patrón de coincidencia en los constructores del tipo de lista:

```
listSum :: [Int] -> Int
listSum []          = 0
listSum (x:xs) = x + listSum xs
```

Podemos hacer coincidir más valores especificando un patrón más elaborado:

```
sumTwoPer :: [Int] -> Int
sumTwoPer [] = 0
sumTwoPer (x1:x2:xs) = x1 + x2 + sumTwoPer xs
sumTwoPer (x:xs) = x + sumTwoPer xs
```

Tenga en cuenta que en el ejemplo anterior, tuvimos que proporcionar una coincidencia de patrón más exhaustiva para manejar los casos en los que se proporciona una lista de longitud impar como argumento.

Haskell Prelude define muchos elementos incorporados para el manejo de listas, como `map` , `filter` , etc. Siempre que sea posible, debe usarlos en lugar de escribir sus propias funciones recursivas.

Accediendo a elementos en listas

Acceda al elemento *n* th de una lista (basado en cero):

```
list = [1 .. 10]

firstElement = list !! 0           -- 1
```

Tenga en cuenta que `!!` Es una función parcial, por lo que ciertas entradas producen errores:

```
list !! (-1)     -- *** Exception: Prelude.!!: negative index

list !! 1000     -- *** Exception: Prelude.!!: index too large
```

También hay `Data.List.genericIndex` , una versión sobrecargada de `!!` , que acepta cualquier valor `Integral` como índice.

```
import Data.List (genericIndex)

list `genericIndex` 4              -- 5
```

Cuando se implementan como listas enlazadas individualmente, estas operaciones llevan tiempo *O (n)* . Si accede con frecuencia a los elementos por índice, probablemente sea mejor usar `Data.Vector` (del paquete vectorial) u otras estructuras de datos.

Gamas

Crear una lista del 1 al 10 es simple usando la notación de rango:

```
[1..10]    -- [1,2,3,4,5,6,7,8,9,10]
```

Para especificar un paso, agregue una coma y el siguiente elemento después del elemento de inicio:

```
[1,3..10]  -- [1,3,5,7,9]
```

Tenga en cuenta que Haskell siempre toma el paso como la diferencia aritmética entre términos, y que no puede especificar más que los primeros dos elementos y el límite superior:

```
[1,3,5..10] -- error
[1,3,9..20] -- error
```

Para generar un rango en orden descendente, especifique siempre el paso negativo:

```
[5..1]     -- []

[5,4..1]   -- [5,4,3,2,1]
```

Debido a que Haskell no es estricto, los elementos de la lista se evalúan solo si son necesarios, lo que nos permite utilizar listas infinitas. `[1..]` es una lista infinita que comienza desde 1. Esta lista se puede vincular a una variable o pasar como un argumento de función:

```
take 5 [1..]   -- returns [1,2,3,4,5] even though [1..] is infinite
```

Tenga cuidado al usar rangos con valores de punto flotante, ya que acepta derrames hasta la mitad delta, para evitar los problemas de redondeo:

```
[1.0,1.5..2.4]    -- [1.0,1.5,2.0,2.5] , though 2.5 > 2.4

[1.0,1.1..1.2]    -- [1.0,1.1,1.2000000000000002] , though 1.2000000000000002 > 1.2
```

Los rangos no solo funcionan con números, sino con cualquier tipo que implemente `Enum` typeclass. Dadas algunas variables enumerables `a` , `b` , `c` , la sintaxis del rango es equivalente a llamar a estos métodos `Enum` :

```
[a..]    == enumFrom a
[a..c]   == enumFromTo a c
[a,b..]  == enumFromThen a b
[a,b..c] == enumFromThenTo a b c
```

Por ejemplo, con `Bool` es

```
 [False ..]      -- [False,True]
```

Observe el espacio después de `False` , para evitar que esto se analiza como un nombre de módulo de clasificación (es decir `False..` se analiza como `.` Desde un módulo `False`).

Funciones básicas en listas

```
head [1..10]       --    1

last [1..20]       --    20

tail [1..5]        --    [2, 3, 4, 5]

init [1..5]        --    [1, 2, 3, 4]

length [1 .. 10]   --    10

reverse [1 .. 10]  --    [10, 9 .. 1]

take 5 [1, 2 .. ]  --    [1, 2, 3, 4, 5]

drop 5 [1 .. 10]   --    [6, 7, 8, 9, 10]

concat [[1,2], [], [4]]   --    [1,2,4]
```

pliegue

Así es como se implementa el pliegue izquierdo. Observe cómo el orden de los argumentos en la función de paso se invierte en comparación con `foldr` (el pliegue derecho):

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []     =  acc
foldl f acc (x:xs) =  foldl f (f acc x) xs         -- = foldl f (acc `f` x) xs
```

El pliegue izquierdo, `foldl` , se asocia a la izquierda. Es decir:

```
foldl (+) 0 [1, 2, 3]     -- is equivalent to ((0 + 1) + 2) + 3
```

La razón es que `foldl` se evalúa de esta manera (mire el paso inductivo de `foldl`):

```
foldl (+) 0 [1, 2, 3]                       --  foldl (+)    0   [ 1,   2,   3 ]
foldl (+) ((+) 0 1) [2, 3]                  --  foldl (+)   (0 + 1)   [ 2,   3 ]
foldl (+) ((+) ((+) 0 1) 2) [3]             --  foldl (+)  ((0 + 1) + 2)   [ 3 ]
foldl (+) ((+) ((+) ((+) 0 1) 2) 3) []      --  foldl (+) (((0 + 1) + 2) + 3) []
((+) ((+) ((+) 0 1) 2) 3)                   --            (((0 + 1) + 2) + 3)
```

La última línea es equivalente a `((0 + 1) + 2) + 3` . Esto se debe a que `(fab)` es lo mismo que `(a `f` b)` en general, y por lo tanto `((+) 0 1)` es lo mismo que `(0 + 1)` en particular.

plegar

Así es como se implementa el pliegue correcto:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)              -- = x `f` foldr f z xs
```

El pliegue derecho, `foldr` , se asocia a la derecha. Es decir:

```
foldr (+) 0 [1, 2, 3]      -- is equivalent to 1 + (2 + (3 + 0))
```

La razón es que `foldr` se evalúa de esta manera (observe el paso inductivo de `foldr`):

```
foldr (+) 0 [1, 2, 3]                       --          foldr (+) 0  [1,2,3]
(+) 1 (foldr (+) 0 [2, 3])                  -- 1 +        foldr (+) 0  [2,3]
(+) 1 ((+) 2 (foldr (+) 0 [3]))             -- 1 + (2 +     foldr (+) 0  [3])
(+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))      -- 1 + (2 + (3 +  foldr (+) 0 []))
(+) 1 ((+) 2 ((+) 3 0))                     -- 1 + (2 + (3 +            0   ))
```

La última línea es equivalente a `1 + (2 + (3 + 0))` , porque `((+) 3 0)` es lo mismo que `(3 + 0)` .

Transformando con `map`

A menudo, deseamos convertir o transformar el contenido de una colección (una lista o algo que

se pueda recorrer). En Haskell usamos el `map` :

```
-- Simple add 1
map (+ 1) [1,2,3]
[2,3,4]

map odd [1,2,3]
[True,False,True]

data Gender = Male | Female deriving Show
data Person = Person String Gender Int deriving Show

-- Extract just the age from a list of people
map (\(Person n g a) -> a) [(Person "Alex" Male 31),(Person "Ellie" Female 29)]
[31,29]
```

Filtrado con `filter`

Dada una lista:

```
li = [1,2,3,4,5]
```

podemos filtrar una lista con un predicado usando `filter :: (a -> Bool) -> [a] -> [a]` :

```
filter (== 1) li       -- [1]

filter (even) li       -- [2,4]

filter (odd) li        -- [1,3,5]

-- Something slightly more complicated
comfy i = notTooLarge && isEven
  where
     notTooLarge = (i + 1) < 5
     isEven = even i

filter comfy li        -- [2]
```

Por supuesto que no se trata solo de números:

```
data Gender = Male | Female deriving Show
data Person = Person String Gender Int deriving Show

onlyLadies :: [Person] -> Person
onlyLadies x = filter isFemale x
  where
    isFemale (Person _ Female _) = True
    isFemale _ = False

onlyLadies [(Person "Alex" Male 31),(Person "Ellie" Female 29)]
-- [Person "Ellie" Female 29]
```

Listas de cierre y descompresión

zip toma dos listas y devuelve una lista de pares correspondientes:

```
zip []     _      = []
zip _      []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs

> zip [1,3,5] [2,4,6]
> [(1,2),(3,4),(5,6)]
```

Comprimiendo dos listas con una función:

```
zipWith f  []     _      = []
zipWith f  _      []     = []
zipWith f  (a:as) (b:bs) = f a b : zipWith f as bs

> zipWith (+) [1,3,5] [2,4,6]
> [3,7,11]
```

Descomprimir una lista:

```
unzip = foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

> unzip [(1,2),(3,4),(5,6)]
> ([1,3,5],[2,4,6])
```

Lea Liza en línea: https://riptutorial.com/es/haskell/topic/2281/liza

Capítulo 43: Los funtores comunes como base de los comonads cofree.

Examples

Cofree Empty ~~ Empty

Dado

```
data Empty a
```

tenemos

```
data Cofree Empty a
   --  = a :< ...  not possible!
```

Cofree (Const c) ~~ escritor c

Dado

```
data Const c a = Const c
```

tenemos

```
data Cofree (Const c) a
     = a :< Const c
```

que es isomorfo para

```
data Writer c a = Writer c a
```

Cofree Identity ~~ Stream

Dado

```
data Identity a = Identity a
```

tenemos

```
data Cofree Identity a
     = a :< Identity (Cofree Identity a)
```

que es isomorfo para

```
data Stream a = Stream a (Stream a)
```

Cofree Maybe ~~ NonEmpty

Dado

```
data Maybe a = Just a
             | Nothing
```

tenemos

```
data Cofree Maybe a
     = a :< Just (Cofree Maybe a)
     | a :< Nothing
```

que es isomorfo para

```
data NonEmpty a
     = NECons a (NonEmpty a)
     | NESingle a
```

Cofree (Writer w) ~~ WriterT w Stream

Dado

```
data Writer w a = Writer w a
```

tenemos

```
data Cofree (Writer w) a
     = a :< (w, Cofree (Writer w) a)
```

que es equivalente a

```
data Stream (w,a)
     = Stream (w,a) (Stream (w,a))
```

que puede escribirse correctamente como `WriterT w Stream` con

```
data WriterT w m a = WriterT (m (w,a))
```

Cofree (Either e) ~~ NonEmptyT (Writer e)

Dado

```
data Either e a = Left e
                | Right a
```

tenemos

```
data Cofree (Either e) a
     = a :< Left e
     | a :< Right (Cofree (Either e) a)
```

que es isomorfo para

```
data Hospitable e a
     = Sorry_AllIHaveIsThis_Here'sWhy a e
     | EatThis a (Hospitable e a)
```

o, si se compromete a solo evaluar el registro después del resultado completo, `NonEmptyT (Writer e) a` con

```
data NonEmptyT (Writer e) a = NonEmptyT (e,a,[a])
```

Cofree (Lector x) ~~ Moore x

Dado

```
data Reader x a = Reader (x -> a)
```

tenemos

```
data Cofree (Reader x) a
     = a :< (x -> Cofree (Reader x) a)
```

que es isomorfo para

```
data Plant x a
     = Plant a (x -> Plant x a)
```

También conocido como máquina de Moore .

Lea Los funtores comunes como base de los comonads cofree. en línea: https://riptutorial.com/es/haskell/topic/8258/los-funtores-comunes-como-base-de-los-comonads-cofree-

Capítulo 44: Mejoramiento

Examples

Compilando su programa para perfilar

El compilador de GHC tiene un soporte maduro para compilar con anotaciones de perfiles.

El uso de los `-prof` y `-fprof-auto` al compilar agregará soporte a su binario para los indicadores de perfil para su uso en tiempo de ejecución.

Supongamos que tenemos este programa:

```
main = print (fib 30)
fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Compilado así

```
ghc -prof -fprof-auto -rtsopts Main.hs
```

Luego lo ejecutó con las opciones del sistema de tiempo de ejecución para el perfil:

```
./Main +RTS -p
```

Veremos un archivo `main.prof` creado después de la ejecución (una vez que el programa haya salido), y esto nos dará todo tipo de información de perfiles, como centros de costos, que nos da un desglose del costo asociado con la ejecución de las diversas partes del código. :

```
    Wed Oct 12 16:14 2011 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -RTS

        total time  =        0.68 secs   (34 ticks @ 20 ms)
        total alloc = 204,677,844 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

fib         Main    100.0  100.0


                                                 individual     inherited
COST CENTRE MODULE                  no.     entries  %time %alloc   %time %alloc

MAIN        MAIN                    102           0    0.0    0.0   100.0  100.0
 CAF        GHC.IO.Handle.FD        128           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv   120           0    0.0    0.0     0.0    0.0
 CAF        GHC.Conc.Signal         110           0    0.0    0.0     0.0    0.0
 CAF        Main                    108           0    0.0    0.0   100.0  100.0
  main      Main                    204           1    0.0    0.0   100.0  100.0
   fib      Main                    205     2692537  100.0  100.0   100.0  100.0
```

Centros de costo

Los centros de costos son anotaciones en un programa de Haskell que pueden ser agregados automáticamente por el compilador de GHC - usando `-fprot-auto` - o por un programador usando `{-# SCC "name" #-} <expression>` , donde "nombre" es cualquier nombre que desee y `<expression>` es cualquier expresión de Haskell válida:

```
-- Main.hs
main :: IO ()
main = do let l = [1..9999999]
          print $ {-# SCC "print_list" #-} (length l)
```

La `-fprof` con `-fprof` y la ejecución con `+RTS -p` por ejemplo, `ghc -prof -rtsopts Main.hs && ./Main.hs +RTS -p` producirían `Main.prof` una vez que el programa haya salido.

Lea Mejoramiento en línea: https://riptutorial.com/es/haskell/topic/4342/mejoramiento

Capítulo 45: Módulos

Sintaxis

- Nombre del módulo donde - exportar todos los nombres declarados en este archivo
- Nombre del módulo (functionOne, Type (..)) donde - exportar solo los constructores de functionOne, Type y Type
- Importar módulo - importar todos los nombres exportados del módulo
- importar módulo calificado como MN - importación calificada
- Módulo de importación (justThisFunction): importa solo ciertos nombres de un módulo
- Importar ocultación del módulo (functionName, Type): importa todos los nombres de un módulo, excepto functionName y Type

Observaciones

Haskell tiene soporte para módulos:

- un módulo puede exportar todos, o un subconjunto de sus tipos de miembros y funciones
- un módulo puede "reexportar" los nombres que importó de otros módulos

En el extremo consumidor de un módulo, uno puede:

- importar todos, o un subconjunto de miembros del módulo
- ocultar las importaciones de un miembro particular o conjunto de miembros

haskell.org tiene un gran capítulo sobre la definición del módulo.

Examples

Definiendo tu propio módulo

Si tenemos un archivo llamado `Business.hs` , podemos definir un módulo de `Business` que se puede `import` , por ejemplo:

```
module Business (
    Person (..), -- ^ Export the Person type and all its constructors and field names
    employees   -- ^ Export the employees function
) where
-- begin types, function definitions, etc
```

Una jerarquía más profunda es, por supuesto, posible; vea el ejemplo de los nombres de módulos jerárquicos .

Constructores exportadores

Para exportar el tipo y todos sus constructores, uno debe usar la siguiente sintaxis:

```
module X (Person (..)) where
```

Entonces, para las siguientes definiciones de nivel superior en un archivo llamado `People.hs` :

```
data Person = Friend String | Foe deriving (Show, Eq, Ord)

isFoe Foe = True
isFoe _   = False
```

Este módulo de declaración en la parte superior:

```
module People (Person (..)) where
```

Solo exportaría `Person` y sus constructores `Friend` y `Foe` .

Si se omite la lista de exportación que sigue a la palabra clave del módulo, se exportarán todos los nombres vinculados en el nivel superior del módulo:

```
module People where
```

Exportaría `Person` , sus constructores y la función `isFoe` .

Importación de miembros específicos de un módulo

Haskell admite la importación de un subconjunto de elementos de un módulo.

```
import qualified Data.Stream (map) as D
```

solo importaría el `map` desde `Data.Stream` , y las llamadas a esta función requerirían `D.` .:

```
D.map odd [1..]
```

de lo contrario, el compilador intentará usar la función de `map Prelude` .

Ocultar Importaciones

Prelude a menudo define funciones cuyos nombres se usan en otros lugares. Si no se ocultan dichas importaciones (o se utilizan importaciones calificadas donde se producen conflictos) se producirán errores de compilación.

`Data.Stream` define funciones denominadas `map` , `head` y `tail` que normalmente chocan con las

definidas en Prelude. Podemos ocultar esas importaciones de Prelude usando la `hiding` :

```
import Data.Stream -- everything from Data.Stream
import Prelude hiding (map, head, tail, scan, foldl, foldr, filter, dropWhile, take) -- etc
```

En realidad, se requeriría demasiado código para ocultar los choques de Prelude de esta manera, por lo que de hecho utilizaría una importación `qualified` de `Data.Stream` en `Data.Stream` lugar.

Importaciones Calificadas

Cuando varios módulos definen las mismas funciones por nombre, el compilador se quejará. En tales casos (o para mejorar la legibilidad), podemos utilizar una importación `qualified` :

```
import qualified Data.Stream as D
```

Ahora podemos evitar los errores del compilador de ambigüedad cuando usamos `map` , que se define en `Prelude` y `Data.Stream` :

```
map (== 1) [1,2,3] -- will use Prelude.map
D.map (odd) (fromList [1..]) -- will use Data.Stream.map
```

También es posible importar un módulo con solo los nombres de conflicto calificados a través de `import Data.Text as T` , lo que permite tener `Text` lugar de `T.Text` etc.

Nombres de módulos jerárquicos

Los nombres de los módulos siguen la estructura jerárquica del sistema de archivos. Con el siguiente diseño de archivo:

```
Foo/
├── Baz/
│   └── Quux.hs
└── Bar.hs
Foo.hs
Bar.hs
```

Los encabezados del módulo se verían así:

```
-- file Foo.hs
module Foo where

-- file Bar.hs
module Bar where

-- file Foo/Bar.hs
module Foo.Bar where

-- file Foo/Baz/Quux.hs
module Foo.Baz.Quux where
```

Tenga en cuenta que:

- El nombre del módulo se basa en la ruta del archivo que declara el módulo.
- Las carpetas pueden compartir un nombre con un módulo, lo que da una estructura de denominación jerárquica natural a los módulos

Lea Módulos en línea: https://riptutorial.com/es/haskell/topic/5234/modulos

Capítulo 46: Mónada del estado

Introducción

Las mónadas estatales son un tipo de mónada que llevan un estado que puede cambiar durante cada ejecución de cálculo en la mónada. Las implementaciones son generalmente de la forma `State sa` que representa un cálculo que lleva y potencialmente modifica un estado de tipo `s` y produce un resultado de tipo `a` , pero el término "mónada de estado" generalmente se refiere a cualquier mónada que lleva un estado. El paquete `mtl` y `transformers` proporciona implementaciones generales de mónadas estatales.

Observaciones

Los recién llegados a Haskell a menudo se alejan de la mónada `State` y la tratan como un tabú, ya que el beneficio declarado de la programación funcional es evitar el estado, así que, ¿no lo pierde cuando usa `State` ? Una vista más matizada es que:

- El estado puede ser útil en *pequeñas dosis controladas* ;
- El tipo de `State` proporciona la capacidad de controlar la dosis con mucha precisión.

Las razones son que si tienes `action :: State sa` , esto te dice que:

- `action` es especial porque depende de un estado;
- El estado tiene el tipo `s` , por lo que la `action` no puede verse influenciada por ningún valor antiguo en su programa; solo se puede acceder a una `s` o algún valor desde algunos `s` ;
- El `runState :: State sa -> s -> (a, s)` pone una "barrera" alrededor de la acción con estado, de modo que su efectividad *no se* puede observar desde fuera de esa barrera.

Por lo tanto, este es un buen conjunto de criterios para utilizar el `State` en un escenario particular. Quiere ver que su código está *minimizando el alcance del estado* , tanto al elegir un tipo estrecho para `s` como al poner `runState` más cerca posible al "fondo" (para que sus acciones puedan verse influidas por tan pocas cosas como posible.

Examples

Numerar los nodos de un árbol con un contador.

Tenemos un tipo de datos de árbol como este:

```
data Tree a = Tree a [Tree a] deriving Show
```

Y deseamos escribir una función que asigne un número a cada nodo del árbol, desde un contador incremental:

```
tag :: Tree a -> Tree (a, Int)
```

El largo camino

Primero lo haremos a lo largo, ya que ilustra muy bien la mecánica de bajo nivel de la mónada `State` .

```
import Control.Monad.State

-- Function that numbers the nodes of a `Tree`.
tag :: Tree a -> Tree (a, Int)
tag tree =
    -- tagStep is where the action happens.  This just gets the ball
    -- rolling, with `0` as the initial counter value.
    evalState (tagStep tree) 0

-- This is one monadic "step" of the calculation.  It assumes that
-- it has access to the current counter value implicitly.
tagStep :: Tree a -> State Int (Tree (a, Int))
tagStep (Tree a subtrees) = do
    -- The `get :: State s s` action accesses the implicit state
    -- parameter of the State monad.  Here we bind that value to
    -- the variable `counter`.
    counter <- get

    -- The `put :: s -> State s ()` sets the implicit state parameter
    -- of the `State` monad.  The next `get` that we execute will see
    -- the value of `counter + 1` (assuming no other puts in between).
    put (counter + 1)

    -- Recurse into the subtrees.  `mapM` is a utility function
    -- for executing a monadic actions (like `tagStep`) on a list of
    -- elements, and producing the list of results.  Each execution of
    -- `tagStep` will be executed with the counter value that resulted
    -- from the previous list element's execution.
    subtrees' <- mapM tagStep subtrees

    return $ Tree (a, counter) subtrees'
```

Refactorización

Dividir el contador en una acción postIncrement

La parte en la que nos encontramos `get` ting el contador actual y luego `put` ting contador + 1 se puede separar en una `postIncrement` acción, similar a lo que muchos lenguajes de tipo C proporcionan:

```
postIncrement :: Enum s => State s s
postIncrement = do
    result <- get
    modify succ
    return result
```

Divide la caminata del árbol en una función de orden superior.

La lógica de la caminata del árbol se puede dividir en su propia función, como esta:

```
mapTreeM :: Monad m => (a -> m b) -> Tree a -> m (Tree b)
mapTreeM action (Tree a subtrees) = do
    a' <- action a
    subtrees' <- mapM (mapTreeM action) subtrees
    return $ Tree a' subtrees'
```

Con esto y la función `postIncrement` podemos reescribir `tagStep` :

```
tagStep :: Tree a -> State Int (Tree (a, Int))
tagStep = mapTreeM step
    where step :: a -> State Int (a, Int)
          step a = do
              counter <- postIncrement
              return (a, counter)
```

Usa la clase `Traversable`

La solución `mapTreeM` anterior se puede reescribir fácilmente en una instancia de la clase `Traversable` :

```
instance Traversable Tree where
    traverse action (Tree a subtrees) =
        Tree <$> action a <*> traverse action subtrees
```

Tenga en cuenta que esto nos obligó a utilizar `Applicative` (el operador `<*>`) en lugar de `Monad` .

Con eso, ahora podemos escribir `tag` como un profesional:

```
tag :: Traversable t => t a -> t (a, Int)
tag init t = evalState (traverse step t) 0
    where step a = do tag <- postIncrement
                      return (a, tag)
```

¡Tenga en cuenta que esto funciona para cualquier tipo de `Traversable` , no solo para nuestro tipo de `Tree` !

Deshacerse de la `Traversable` desplazamiento de `Traversable`

GHC tiene una extensión `DeriveTraversable` que elimina la necesidad de escribir la instancia anterior:

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable, DeriveTraversable #-}
```

```
data Tree a = Tree a [Tree a]
            deriving (Show, Functor, Foldable, Traversable)
```

Lea Mónada del estado en línea: https://riptutorial.com/es/haskell/topic/5740/monada-del-estado

Capítulo 47: Mónadas

Introducción

Una mónada es un tipo de datos de acciones compuestas. `Monad` es la clase de constructores de tipo cuyos valores representan tales acciones. Tal vez `IO` es el más reconocible uno: un valor de `IO a` es una "receta para recuperar un `a` valor en el mundo real".

Decimos que un constructor de tipo `m` (como `[]` o `Maybe`) *forma una mónada* si hay una `instance Monad m` cumple ciertas leyes sobre la composición de las acciones. Entonces podemos razonar sobre `ma` como una "acción cuyo resultado tiene el tipo `a` ".

Examples

La mónada Tal vez

`Maybe` se utiliza para representar valores posiblemente vacíos, similar a `null` en otros idiomas. Normalmente se usa como el tipo de salida de funciones que pueden fallar de alguna manera.

Considera la siguiente función:

```
halve :: Int -> Maybe Int
halve x
  | even x = Just (x `div` 2)
  | odd x  = Nothing
```

Piense en la `halve` como una acción, dependiendo de un `Int` , que intenta reducir a la mitad el número entero, fallando si es impar.

¿Cómo `halve` un número entero tres veces?

```
takeOneEighth :: Int -> Maybe Int            -- (after you read the 'do' sub-section:)
takeOneEighth x =
  case halve x of                               --  do {
    Nothing -> Nothing
    Just oneHalf ->                             --      oneHalf    <- halve x
      case halve oneHalf of
        Nothing -> Nothing
        Just oneQuarter ->                      --      oneQuarter <- halve oneHalf
          case halve oneQuarter of
            Nothing -> Nothing                  --      oneEighth  <- halve oneQuarter
            Just oneEighth ->
              Just oneEighth                    --      return oneEighth }
```

- `takeOneEighth` es una *secuencia* de tres pasos de la `halve` encadenados.
- Si un paso de la `halve` falla, queremos que la composición completa `takeOneEighth` falle.
- Si un paso de la `halve` tiene éxito, queremos canalizar su resultado hacia adelante.

```
instance Monad Maybe where
  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= f  = Nothing                          -- infixl 1 >>=
  Just x  >>= f  = Just (f x)                       -- also, f =<< m = m >>= f

  -- return :: a -> Maybe a
  return x       = Just x
```

y ahora podemos escribir:

```
takeOneEighth :: Int -> Maybe Int
takeOneEighth x = halve x >>= halve >>= halve             -- or,
    -- return x >>= halve >>= halve >>= halve             -- which is parsed as
    -- (((return x) >>= halve) >>= halve) >>= halve       -- which can also be written as
    -- (halve =<<) . (halve =<<) . (halve =<<) $ return x    -- or, equivalently, as
    --  halve <=<     halve <=<     halve      $        x
```

La composición de Kleisli `<=<` se define como `(g <=< f) x = g =<< fx` , o equivalentemente como `(f >=> g) x = fx >>= g` . Con ello la definición anterior se convierte en justa.

```
takeOneEighth :: Int -> Maybe Int
takeOneEighth = halve <=< halve <=< halve              -- infixr 1 <=<
        -- or, equivalently,
        --      halve >=> halve >=> halve              -- infixr 1 >=>
```

Hay tres leyes de la mónada que deben ser obedecidas por cada mónada, es decir, cada tipo que es una instancia de la clase de tipos de la `Monad` :

```
1.  return x >>= f  =  f x
2.    m >>= return  =  m
3. (m >>= g) >>= h  =  m >>= (\y -> g y >>= h)
```

donde `m` es una mónada, `f` tiene tipo `a -> mb` y `g` tiene tipo `b -> mc` .

O de manera equivalente, utilizando el operador de composición Kleisli `>=>` definido anteriormente:

```
1.    return >=> g  =  g                   -- do { y <- return x ; g y } == g x
2.    f >=> return  =  f                   -- do { y <- f x ; return y } == f x
3. (f >=> g) >=> h  =  f >=> (g >=> h)     -- do { z <- do { y <- f x; g y } ; h z }
                                           --  == do { y <- f x ; do { z <- g y; h z } }
```

Obedecer estas leyes hace que sea mucho más fácil razonar acerca de la mónada, porque garantiza que el uso de funciones monádicas y su composición se comporte de una manera razonable, similar a otras mónadas.

Vamos a comprobar si la mónada `Maybe` obedece las tres leyes de la mónada.

1. **La ley de identidad de la izquierda -** `return x >>= f = fx`

```
return z >>= f
```

```
= (Just z) >>= f
= f z
```

2. **La ley de identidad correcta** - `m >>= return = m`

- `Just` constructor de datos

```
Just z >>= return
= return z
= Just z
```

- Constructor de `Nothing` datos

```
Nothing >>= return
= Nothing
```

3. **La ley de asociatividad** - `(m >>= f) >>= g = m >>= (\x -> fx >>= g)`

- `Just` constructor de datos

```
-- Left-hand side
((Just z) >>= f) >>= g
= f z >>= g

-- Right-hand side
(Just z) >>= (\x -> f x >>= g)
(\x -> f x >>= g) z
= f z >>= g
```

- Constructor de `Nothing` datos

```
-- Left-hand side
(Nothing >>= f) >>= g
= Nothing >>= g
= Nothing

-- Right-hand side
Nothing >>= (\x -> f x >>= g)
= Nothing
```

IO mónada

No hay forma de obtener un valor de tipo `a` fuera de una expresión de tipo `IO a` y no debería haber. Esto es en realidad una gran parte de por qué las mónadas se utilizan para modelar `IO` .

Se puede considerar que una expresión de tipo `IO a` representa una acción que puede interactuar con el mundo real y, si se ejecuta, daría lugar a algo de tipo `a` . Por ejemplo, la función `getLine :: IO String` del preludio no significa que debajo de `getLine` haya una cadena específica que pueda extraer; significa que `getLine` representa la acción de obtener una línea desde la entrada estándar.

No es sorprendente que `main :: IO ()` ya que un programa Haskell representa una computación /

acción que interactúa con el mundo real.

Lo que *puede* hacer con expresiones de tipo `IO a` porque `IO` es una mónada:

- Secuencia dos acciones usando `(>>)` para producir una nueva acción que ejecuta la primera acción, descarta cualquier valor que produzca y luego ejecuta la segunda acción.

```
-- print the lines "Hello" then "World" to stdout
putStrLn "Hello" >> putStrLn "World"
```

- A veces no desea descartar el valor que se produjo en la primera acción; en realidad, le gustaría que se introdujera en una segunda acción. Para eso, tenemos `>>=` . Para `IO` , tiene tipo `(>>=) :: IO a -> (a -> IO b) -> IO b` .

```
-- get a line from stdin and print it back out
getLine >>= putStrLn
```

- Tome un valor normal y conviértalo en una acción que inmediatamente devuelva el valor que le dio. Esta función es menos útil, obviamente, hasta que empiece a usar notación `do`

```
-- make an action that just returns 5
return 5
```

Más de la Wiki de Haskell en la mónada IO aquí .

Lista Mónada

Las listas forman una mónada. Tienen una instanciación de mónada equivalente a esta:

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
```

Podemos usarlos para emular el no determinismo en nuestros cálculos. Cuando usamos `xs >>= f` , la función `f :: a -> [b]` se mapea sobre la lista `xs` , obteniendo una lista de listas de resultados de cada aplicación de `f` sobre cada elemento de `xs` , y todas las listas de los resultados se concatenan en una lista de todos los resultados. Como ejemplo, calculamos una suma de dos números no deterministas utilizando **do-notation** , representada por la lista de sumas de todos los pares de enteros de dos listas, cada una de las cuales representa todos los valores posibles de un número no determinista:

```
sumnd xs ys = do
  x <- xs
  y <- ys
  return (x + y)
```

O equivalentemente, usando `liftM2` en `Control.Monad` :

```
sumnd = liftM2 (+)
```

obtenemos:

```
> sumnd [1,2,3] [0,10]
[1,11,2,12,3,13]
```

La mónada como subclase de aplicativo

A partir de GHC 7.10, el `Applicative` es una superclase de la `Monad` (es decir, todo tipo que sea una `Monad` también debe ser un `Applicative`). Todos los métodos de `Applicative` (`pure` , `<*>`) se pueden implementar en términos de métodos de `Monad` (`return` , `>>=`).

Es obvio que los propósitos `pure` y de `return` son equivalentes, tan `pure = return` . La definición de `<*>` es demasiado clara:

```
mf <*> mx = do { f <- mf; x <- mx; return (f x) }
       -- = mf >>= (\f -> mx >>= (\x -> return (f x)))
       -- = [r   | f <- mf, x <- mx, r <- return (f x)]   -- with MonadComprehensions
       -- = [f x | f <- mf, x <- mx]
```

Esta función se define como `ap` en las bibliotecas estándar.

Por lo tanto, si ya ha definido una instancia de `Monad` para un tipo, efectivamente puede obtener una instancia de `Applicative` para ella "gratis" definiendo

```
instance Applicative < type > where
    pure  = return
    (<*>) = ap
```

Al igual que con las leyes de la mónada, estas equivalencias no se aplican, pero los desarrolladores deben asegurarse de que siempre se respeten.

No hay forma general de extraer valor de un cálculo monádico

Puede ajustar valores en acciones y canalizar el resultado de un cálculo en otro:

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

Sin embargo, la definición de una Mónada no garantiza la existencia de una función de tipo `Monad m => ma -> a` .

Eso significa que, en general, **no hay forma de extraer un valor de un cálculo** (es decir, "desenvolverlo"). Este es el caso de muchos casos:

```
extract :: Maybe a -> a
extract (Just x) = x          -- Sure, this works, but...
extract Nothing  = undefined  -- We can’t extract a value from failure.
```

Específicamente, no hay ninguna función `IO a -> a` , que a menudo confunde a los principiantes;

ver este ejemplo

hacer notación

`do` anotación es azúcar sintáctica para las mónadas. Estas son las reglas:

```
do x <- mx                              do x <- mx
   y <- my       is equivalent to           do y <- my
   ...                                         ...
```

```
do let a = b                            let a = b in
   ...           is equivalent to           do ...
```

```
do m                                    m >> (
   e             is equivalent to           e)
```

```
do x <- m                               m >>= (\x ->
   e             is equivalent to           e)
```

```
do m             is equivalent to       m
```

Por ejemplo, estas definiciones son equivalentes:

```
example :: IO Integer
example =
  putStrLn "What's your name?" >> (
    getLine >>= (\name ->
      putStrLn ("Hello, " ++ name ++ ".") >> (
        putStrLn "What should we return?" >> (
          getLine >>= (\line ->
            let n = (read line :: Integer) in
              return (n + n))))))
```

```
example :: IO Integer
example = do
  putStrLn "What's your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ ".")
  putStrLn "What should we return?"
  line <- getLine
  let n = (read line :: Integer)
  return (n + n)
```

Definición de mónada

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

La función más importante para tratar con las mónadas es el **operador de enlace** `>>=` :

```
(>>=) :: m a -> (a -> m b) -> m b
```

- Piense en `ma` como *"una acción con un `a` resultado".*
- Piense en `a -> mb` *"una acción (en función de una* como `a` *parámetro) con una `b` . Consecuencia".*

`>>=` **secuencia dos acciones juntas canalizando el resultado de la primera acción a la segunda.**

La otra función definida por `Monad` es:

```
return :: a -> m a
```

Su nombre es desafortunado: este `return` no tiene nada que ver con la palabra clave de `return` encontrada en lenguajes de programación imperativos.

`return x` **es la acción trivial que produce `x` como su resultado.** (Es trivial en el siguiente sentido :)

```
return x >>= f       ≡  f x     --  "left identity" monad law
       x >>= return  ≡  x       -- "right identity" monad law
```

Lea Mónadas en línea: https://riptutorial.com/es/haskell/topic/2968/monadas

Capítulo 48: Mónadas comunes como mónadas libres.

Examples

Libre Vacío ~~ Identidad

Dado

```
data Empty a
```

tenemos

```
data Free Empty a
     = Pure a
-- the Free constructor is impossible!
```

que es isomorfo para

```
data Identity a
     = Identity a
```

Identidad libre ~~ (Nat,) ~~ escritor Nat

Dado

```
data Identity a = Identity a
```

tenemos

```
data Free Identity a
     = Pure a
     | Free (Identity (Free Identity a))
```

que es isomorfo para

```
data Deferred a
     = Now a
     | Later (Deferred a)
```

o de manera equivalente (si prometes evaluar primero el primer elemento) `(Nat, a)` , también conocido como `Writer Nat a` , con

```
data Nat = Z | S Nat
data Writer Nat a = Writer Nat a
```

Libre Tal vez ~~ MaybeT (Escritor Nat)

Dado

```
data Maybe a = Just a
             | Nothing
```

tenemos

```
data Free Maybe a
     = Pure a
     | Free (Just (Free Maybe a))
     | Free Nothing
```

que es equivalente a

```
data Hopes a
     = Confirmed a
     | Possible (Hopes a)
     | Failed
```

o de manera equivalente (si prometes evaluar primero el primer elemento) `(Nat, Maybe a)` , también `MaybeT (Writer Nat) a` como `MaybeT (Writer Nat) a` con

```
data Nat = Z | S Nat
data Writer Nat a = Writer Nat a
data MaybeT (Writer Nat) a = MaybeT (Nat, Maybe a)
```

Libre (escritor w) ~~ escritor [w]

Dado

```
data Writer w a = Writer w a
```

tenemos

```
data Free (Writer w) a
     = Pure a
     | Free (Writer w (Free (Writer w) a))
```

que es isomorfo para

```
data ProgLog w a
     = Done a
     | After w (ProgLog w a)
```

o, de manera equivalente, (si se compromete a evaluar el registro primero), `Writer [w] a` .

Libre (Const c) ~~ O bien c

Dado

```
data Const c a = Const c
```

tenemos

```
data Free (Const c) a
     = Pure a
     | Free (Const c)
```

que es isomorfo para

```
data Either c a
     = Right a
     | Left c
```

Gratis (Reader x) ~~ Reader (Stream x)

Dado

```
data Reader x a = Reader (x -> a)
```

tenemos

```
data Free (Reader x) a
     = Pure a
     | Free (x -> Free (Reader x) a)
```

que es isomorfo para

```
data Demand x a
     = Satisfied a
     | Hungry (x -> Demand x a)
```

o equivalentemente `Stream x -> a` con

```
data Stream x = Stream x (Stream x)
```

Lea Mónadas comunes como mónadas libres. en línea:
https://riptutorial.com/es/haskell/topic/8256/monadas-comunes-como-monadas-libres-

Capítulo 49: Mónadas Libres

Examples

Las mónadas libres dividen los cálculos monádicos en estructuras de datos e intérpretes

Por ejemplo, un cálculo que involucra comandos para leer y escribir desde el indicador:

Primero describimos los "comandos" de nuestro cálculo como un tipo de datos de Functor

```
{-# LANGUAGE DeriveFunctor #-}

data TeletypeF next
    = PrintLine String next
    | ReadLine (String -> next)
    deriving Functor
```

Luego usamos `Free` para crear la "Monad libre sobre `TeletypeF` " y construir algunas operaciones básicas.

```
import Control.Monad.Free (Free, liftF, iterM)

type Teletype = Free TeletypeF

printLine :: String -> Teletype ()
printLine str = liftF (PrintLine str ())

readLine :: Teletype String
readLine = liftF (ReadLine id)
```

Dado que `Free f` es una `Monad` siempre que `f` es un `Functor` , podemos usar los combinadores de `Monad` estándar (incluida la notación `do`) para crear cálculos de `Teletype` .

```
import Control.Monad -- we can use the standard combinators

echo :: Teletype ()
echo = readLine >>= printLine

mockingbird :: Teletype a
mockingbird = forever echo
```

Finalmente, escribimos un "intérprete" que convierte a `Teletype a` valores en algo que sabemos cómo trabajar como `IO a`

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype = foldFree run where
  run :: TeletypeF a -> IO a
  run (PrintLine str x) = putStrLn *> return x
  run (ReadLine f) = fmap f getLine
```

Que podemos usar para "ejecutar" el cálculo del `Teletype a` en `IO`

```
> interpretTeletype mockingbird
hello
hello
goodbye
goodbye
this will go on forever
this will go on forever
```

Las mónadas libres son como puntos fijos.

Compara la definición de `Free` con la de `Fix` :

```
data Free f a = Return a
              | Free (f (Free f a))

newtype Fix f = Fix { unFix :: f (Fix f) }
```

En particular, compare el tipo del constructor `Free` con el tipo del constructor `Fix` . `Free` coloca un funtor como `Fix` , excepto que `Free` tiene un `Return a` case adicional.

¿Cómo funcionan foldFree y iterM?

Existen algunas funciones para ayudar a eliminar los cálculos de `Free` interpretándolos en otra mónada `m` : `iterM :: (Functor f, Monad m) => (f (ma) -> ma) -> (Free fa -> ma)` y `foldFree :: Monad m => (forall x. fx -> mx) -> (Free fa -> ma)` . ¿Qué están haciendo?

Primero veamos qué se necesitaría para derribar e interpretar manualmente `Teletype a` función de `Teletype a` en `IO` . Podemos ver a `Free fa` como siendo definido

```
data Free f a
    = Pure a
    | Free (f (Free f a))
```

El caso `Pure` es fácil:

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype (Pure x) = return x
interpretTeletype (Free teletypeF) = _
```

Ahora, ¿cómo interpretar un cálculo de `Teletype` que se construyó con el constructor `Free` ? Nos gustaría llegar a un valor de tipo `IO a` mediante el examen de `teletypeF :: TeletypeF (Teletype a)` . Para empezar, escribiremos una función `runIO :: TeletypeF a -> IO a` que asigna una sola capa de la mónada libre a una acción `IO` :

```
runIO :: TeletypeF a -> IO a
runIO (PrintLine msg x) = putStrLn msg *> return x
runIO (ReadLine k) = fmap k getLine
```

Ahora podemos usar `runIO` para completar el resto de `interpretTeletype` . Recuerde que `teletypeF :: TeletypeF (Teletype a)` es una capa del functor `TeletypeF` que contiene el resto de la computación `Free` . Usaremos `runIO` para interpretar la capa más externa (por lo tanto, tenemos `runIO teletypeF :: IO (Teletype a)`) y luego usaremos el combinador `>>=` la mónada `IO` para interpretar el `Teletype a` devuelto `Teletype a` .

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype (Pure x) = return x
interpretTeletype (Free teletypeF) = runIO teletypeF >>= interpretTeletype
```

La definición de `foldFree` es solo la de `interpretTeletype` , excepto que la función `runIO` ha sido eliminada. Como resultado, `foldFree` funciona independientemente de cualquier funtor base particular y de cualquier mónada objetivo.

```
foldFree :: Monad m => (forall x. f x -> m x) -> Free f a -> m a
foldFree eta (Pure x) = return x
foldFree eta (Free fa) = eta fa >>= foldFree eta
```

`foldFree` tiene un tipo de rango 2: `eta` es una transformación natural. Podríamos haber dado a `foldFree` un tipo de `Monad m => (f (Free fa) -> m (Free fa)) -> Free fa -> ma` , pero eso da a `eta` la libertad de inspeccionar el cálculo de `Free` dentro de la capa `f` . Darle a `foldFree` este tipo más restrictivo garantiza que `eta` solo pueda procesar una capa a la vez.

`iterM` le da a la función de plegado la capacidad de examinar la subcomputación. El resultado (monádico) de la iteración anterior está disponible para el siguiente, dentro del parámetro `f` . `iterM` es análogo a un *paramorfismo,* mientras que `foldFree` es como un *catamorfismo* .

```
iterM :: (Monad m, Functor f) => (f (m a) -> m a) -> Free f a -> m a
iterM phi (Pure x) = return x
iterM phi (Free fa) = phi (fmap (iterM phi) fa)
```

La mónada Freer

Hay una formulación alternativa de la mónada libre llamada mónada Freer (o Prompt, u Operational). La mónada Freer no requiere una instancia de Functor para su conjunto de instrucciones subyacentes, y tiene una estructura similar a una lista más reconocible que la mónada estándar gratuita.

La mónada Freer representa programas como una secuencia de *instrucciones* atómicas *que* pertenecen al conjunto de instrucciones `i :: * -> *` . Cada instrucción usa su parámetro para declarar su tipo de retorno. Por ejemplo, el conjunto de instrucciones básicas para la mónada `State` es el siguiente:

```
data StateI s a where
    Get :: StateI s s  -- the Get instruction returns a value of type 's'
    Put :: s -> StateI s ()  -- the Put instruction contains an 's' as an argument and returns
()
```

La secuenciación de estas instrucciones se realiza con `:>>=` constructor. `:>>=` toma una sola

instrucción que devuelve una `a` y la precede al resto del programa, canalizando su valor de retorno a la continuación. En otras palabras, dada una instrucción que devuelve una `a` , y una función para convertir una `a` en un programa que devuelve una `b` `:>>=` producirá un programa que devuelve una `b` .

```
data Freer i a where
    Return :: a -> Freer i a
    (:>>=) :: i a -> (a -> Freer i b) -> Freer i b
```

Tenga en cuenta que `a` se cuantifica existencialmente en `:>>=` constructor. La única manera para que un intérprete para aprender lo que `a` es es por coincidencia de patrones en el GADT `i` .

> **Aparte** : el lema co-Yoneda nos dice que `Freer` es isomorfo a `Free` . Recordemos la definición del funtor de `CoYoneda` :
>
> ```
> data CoYoneda i b where
>   CoYoneda :: i a -> (a -> b) -> CoYoneda i b
> ```
>
> `Freer i` es equivalente a `Free (CoYoneda i)` . Si tomas los constructores de `Free` y configuras `f ~ CoYoneda i` , obtienes:
>
> ```
> Pure :: a -> Free (CoYoneda i) a
> Free :: CoYoneda i (Free (CoYoneda i) b) -> Free (CoYonda i) b ~
>         i a -> (a -> Free (CoYoneda i) b) -> Free (CoYoneda i) b
> ```
>
> a partir del cual podemos recuperar `Freer i` constructores 's por sólo la creación de `Freer i ~ Free (CoYoneda i)` .

Debido `CoYoneda i` es un `Functor` para cualquier `i` , `Freer` es una `Monad` para cualquier `i` , aunque `i` no es un `Functor` .

```
instance Monad (Freer i) where
    return = Return
    Return x >>= f = f x
    (i :>>= g) >>= f = i :>>= fmap (>>= f) g  -- using `(->) r`'s instance of Functor, so fmap
= (.)
```

Los intérpretes se pueden construir para `Freer` asignando instrucciones a alguna mónada de manejador.

```
foldFreer :: Monad m => (forall x. i x -> m x) -> Freer i a -> m a
foldFreer eta (Return x) = return x
foldFreer eta (i :>>= f) = eta i >>= (foldFreer eta . f)
```

Por ejemplo, podemos interpretar la mónada `Freer (StateI s)` utilizando la mónada `State s` regular como un controlador:

```
runFreerState :: Freer (StateI s) a -> s -> (a, s)
runFreerState = State.runState . foldFreer toState
    where toState :: StateI s a -> State s a
```

```
        toState Get = State.get
        toState (Put x) = State.put x
```

Lea Mónadas Libres en línea: https://riptutorial.com/es/haskell/topic/1290/monadas-libres

Capítulo 50: Monoide

Examples

Un ejemplar de monoide para listas.

```
instance Monoid [a] where
    mempty  = []
    mappend = (++)
```

Comprobando las leyes de `Monoid` para esta instancia:

```
mempty `mappend` x = x   <->   [] ++ xs = xs  -- prepending an empty list is a no-op

x `mappend` mempty = x   <->   xs ++ [] = xs  -- appending an empty list is a no-op

x `mappend` (y `mappend` z) = (x `mappend` y) `mappend` z
    <->
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs           -- appending lists is associative
```

Contraer una lista de Monoids en un solo valor

`mconcat :: [a] -> a` es otro método de la `Monoid` clase de tipos :

```
ghci> mconcat [Sum 1, Sum 2, Sum 3]
Sum {getSum = 6}
ghci> mconcat ["concat", "enate"]
"concatenate"
```

Su definición por defecto es `mconcat = foldr mappend mempty` .

Monoides Numéricos

Los números son monoidales de dos maneras: *suma* con 0 como unidad y *multiplicación* con 1 como unidad. Ambos son igualmente válidos y útiles en diferentes circunstancias. Entonces, en lugar de elegir una instancia preferida para los números, hay dos `newtypes` , `Sum` y `Product` para etiquetarlos para la funcionalidad diferente.

```
newtype Sum n = Sum { getSum :: n }

instance Num n => Monoid (Sum n) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)

newtype Product n = Product { getProduct :: n }

instance Num n => Monoid (Product n) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

Esto le permite al desarrollador elegir qué funcionalidad usar envolviendo el valor en el `newtype` apropiado.

```
Sum 3     <> Sum 5     == Sum 8
Product 3 <> Product 5 == Product 15
```

Una instancia de monoide para ()

`()` es un `Monoid` . Dado que solo hay un valor de type `()` , solo hay una cosa que `mempty` y `mappend` podrían hacer:

```
instance Monoid () where
    mempty = ()
    () `mappend` () = ()
```

Lea Monoide en línea: https://riptutorial.com/es/haskell/topic/2211/monoide

Capítulo 51: Operadores de infijo

Observaciones

La mayoría de las funciones de Haskell se llaman con el nombre de la función seguido de argumentos (notación de prefijo). Para funciones que aceptan dos argumentos como (+), a veces tiene sentido proporcionar un argumento antes y después de la función (infijo).

Examples

Preludio

Lógico

`&&` es lógico AND, `||` es lógico o.

`==` es igualdad, `/=` no igualdad, `<` / `<=` menor y `>` / `>=` operadores mayores.

Operadores aritméticos

Los operadores numéricos `+` , `-` y `/` comportan en gran medida como cabría esperar. (División funciona sólo en los números fraccionarios para evitar problemas de redondeo - división entera debe hacerse con `quot` o `div`). Más inusuales son los tres operadores de exponenciación de Haskell:

- `^` toma una base de cualquier tipo de número a una potencia integral no negativa. Esto funciona simplemente por multiplicación iterada (rápida). P.ej

  ```
  4^5  ≡  (4*4)*(4*4)*4
  ```

- `^^` hace lo mismo en el caso positivo, pero también funciona para exponentes negativos. P.ej

  ```
  3^^(-2)  ≡  1 / (2*2)
  ```

  A diferencia de `^` , esto requiere un tipo de base fraccional (es decir, `4^^5 :: Int` no funcionará, solo `4^5 :: Int` o `4^^5 :: Rational`).

- `**` Implementa exponenciación de números reales. Esto funciona para argumentos muy generales, pero es mucho más caro que `^` o `^^` , y generalmente incurre en pequeños errores de punto flotante.

```
2**pi  ≡  exp (pi * log 2)
```

Liza

Hay dos operadores de concatenación:

- `:` (se pronuncia contras) antepone un solo argumento antes de una lista. Este operador es en realidad un constructor y, por lo tanto, también puede usarse para hacer un *patrón de coincidencia* ("construcción inversa") de una lista.
- `++` concatena listas enteras.

```
[1,2] ++ [3,4]  ≡  1 : 2 : [3,4]  ≡  1 : [2,3,4]  ≡  [1,2,3,4]
```

`!!` Es un operador de indexación.

```
[0, 10, 20, 30, 40] !! 3  ≡  30
```

Tenga en cuenta que las listas de indexación son ineficientes (complejidad *O* (*n*) en lugar de *O* (1) para matrices o *O* (log *n*) para mapas); en Haskell generalmente se prefiere deconstruir las listas doblando la coincidencia de patrones en lugar de indexar.

Flujo de control

- `$` es un operador de aplicación de función.

  ```
  f $ x  ≡  f x
         ≡  f(x)  -- disapproved style
  ```

  Este operador se utiliza principalmente para evitar paréntesis. ¡También tiene una versión estricta `$!` , lo que obliga a evaluar el argumento antes de aplicar la función.

- `.` compone funciones

  ```
  (f . g) x  ≡  f (g x)  ≡  f $ g x
  ```

- `>>` secuencias monádicas de acciones. Por ejemplo, `writeFile "foo.txt" "bla" >> putStrLn "Done."` primero escribirá en un archivo, luego imprimirá un mensaje en la pantalla.
- `>>=` hace lo mismo, mientras que también acepta un argumento para pasar de la primera acción a la siguiente. `readLn >>= \x -> print (x^2)` esperará a que el usuario ingrese un número y luego envíe el cuadrado de ese número a la pantalla.

Operadores personalizados

En Haskell, puedes definir cualquier operador de infijo que te guste. Por ejemplo, podría definir el operador que envuelve la lista como

```
(>+<) :: [a] -> [a] -> [a]
env >+< l = env ++ l ++ env

GHCi> "**">+<"emphasis"
"**emphasis**"
```

Siempre debe dar a dichos operadores una declaración de fijeza , como

```
infixr 5 >+<
```

(lo que significaría que `>+<` enlaza tan fuertemente como `++` y `:` do).

Encontrar información sobre operadores de infijo.

Debido a que los infijos son tan comunes en Haskell, regularmente deberá buscar su firma, etc. Afortunadamente, esto es tan fácil como para cualquier otra función:

- Los motores de búsqueda de Haskell Hayoo y Hoogle se pueden usar para operadores de infijo, como para cualquier otra cosa que esté definida en alguna biblioteca.
- En GHCi o IHaskell, puede usar las directivas `:i` y `:t` (**i** nfo y **t** ype) para conocer las propiedades básicas de un operador. Por ejemplo,

  ```
  Prelude> :i +
  class Num a where
    (+) :: a -> a -> a
    ...
        -- Defined in ‘GHC.Num’
  infixl 6 +
  Prelude> :i ^^
  (^^) :: (Fractional a, Integral b) => a -> b -> a
        -- Defined in ‘GHC.Real’
  infixr 8 ^^
  ```

  Esto me dice que `^^` une más fuertemente que `+` , ambos toman tipos numéricos como sus elementos, pero `^^` requiere que el exponente sea integral y la base sea fraccionaria.
  El menos detallado `:t` requiere el operador entre paréntesis, como

  ```
  Prelude> :t (==)
  (==) :: Eq a => a -> a -> Bool
  ```

Lea Operadores de infijo en línea: https://riptutorial.com/es/haskell/topic/6792/operadores-de-infijo

Capítulo 52: Papel

Introducción

La extensión de lenguaje `TypeFamilies` permite al programador definir funciones de nivel de tipo. Lo que distingue a las funciones de tipo de los constructores de tipo no GADT es que los parámetros de las funciones de tipo pueden ser no paramétricos, mientras que los parámetros de los constructores de tipo siempre son paramétricos. Esta distinción es importante para la corrección de la extensión `GeneralizedNewTypeDeriving` . Para explicar esta distinción, los roles se introducen en Haskell.

Observaciones

Véase también `SafeNewtypeDeriving` .

Examples

Papel nominal

Haskell Wiki tiene un ejemplo de un parámetro no paramétrico de una función de tipo:

```
type family Inspect x
type instance Inspect Age = Int
type instance Inspect Int = Bool
```

Aquí `x` no es paramétrico porque para determinar el resultado de aplicar `Inspect` a un argumento de tipo, la función de tipo debe inspeccionar `x` .

En este caso, el papel de `x` es nominal. Podemos declarar el rol explícitamente con la extensión `RoleAnnotations` :

```
type role Inspect nominal
```

Papel representativo

Un ejemplo de un parámetro paramétrico de una función de tipo:

```
data List a = Nil | Cons a (List a)

type family DoNotInspect x
type instance DoNotInspect x = List x
```

Aquí `x` es paramétrico porque para determinar el resultado de aplicar `DoNotInspect` a un argumento de tipo, la función de tipo no necesita inspeccionar `x` .

En este caso, el papel de x es representativo. Podemos declarar el rol explícitamente con la extensión `RoleAnnotations` :

```
type role DoNotInspect representational
```

Papel fantasma

Un parámetro de tipo fantasma tiene un rol fantasma. Los roles fantasmas no pueden ser declarados explícitamente.

Lea Papel en línea: https://riptutorial.com/es/haskell/topic/8753/papel

Capítulo 53: Paralelismo

Parámetros

Tipo / Función	Detalle
`data Eval a`	Eval es una mónada que facilita la definición de estrategias paralelas.
`type Strategy a = a -> Eval a`	Una función que encarna una estrategia de evaluación paralela. La función atraviesa (parte de) su argumento, evaluando subexpresiones en paralelo o en secuencia
`rpar :: Strategy a`	Chispea su argumento (para su evaluación en paralelo)
`rseq :: Strategy a`	Evalúa su argumento a la forma normal de cabeza débil.
`force :: NFData a => a -> a`	evalúa toda la estructura de su argumento, reduciéndolo a su forma normal, antes de devolver el argumento en sí. Es proporcionado por el módulo Control.DeepSeq

Observaciones

El libro de Simon Marlow , Programación concurrente y paralela en Haskell, es sobresaliente y abarca una multitud de conceptos. También es muy accesible incluso para el programador de Haskell más nuevo. Es altamente recomendable y está disponible en PDF o en línea de forma gratuita.

Paralelo vs Concurrente

Simon Marlow lo pone mejor :

> Un programa paralelo es uno que utiliza una multiplicidad de hardware computacional (por ejemplo, varios núcleos de procesador) para realizar un cálculo más rápidamente. El objetivo es llegar a la respuesta anterior, delegando diferentes partes de la computación a diferentes procesadores que se ejecutan al mismo tiempo.
>
> Por el contrario, la concurrencia es una técnica de estructuración de programas en la que hay múltiples hilos de control. Conceptualmente, los hilos de control se ejecutan "al mismo tiempo"; Es decir, el usuario ve sus efectos intercalados. Si realmente se ejecutan al mismo tiempo o no es un detalle de implementación; un programa concurrente puede ejecutarse en un solo procesador a través de la ejecución intercalada o en múltiples procesadores físicos.

Forma normal de la cabeza débil

Es importante ser consciente de cómo funciona la evaluación perezosa. La primera sección de este capítulo brindará una sólida introducción a WHNF y cómo esto se relaciona con la programación paralela y concurrente.

Examples

La Mónada Eval

El paralelismo en Haskell se puede expresar usando la Mónada `Eval` de `Control.Parallel.Strategies` , usando las funciones `rpar` y `rseq` (entre otras).

```
f1 :: [Int]
f1 = [1..100000000]

f2 :: [Int]
f2 = [1..200000000]

main = runEval $ do
  a <- rpar (f1) -- this'll take a while...
  b <- rpar (f2) -- this'll take a while and then some...
  return (a,b)
```

Ejecutar `main` anterior ejecutará y "regresará" inmediatamente, mientras que los dos valores, `a` y `b` se computan en segundo plano a través de `rpar` .

Nota: asegúrese de compilar con `-threaded` para que se produzca la ejecución paralela.

rpar

`rpar :: Strategy a` ejecuta la estrategia dada (recuerda: `type Strategy a = a -> Eval a`) en paralelo:

```
import Control.Concurrent
import Control.DeepSeq
import Control.Parallel.Strategies
import Data.List.Ordered

main = loop
  where
    loop = do
      putStrLn "Enter a number"
      n <- getLine

      let lim = read n :: Int
          hf  = quot lim 2
          result = runEval $ do
            -- we split the computation in half, so we can concurrently calculate primes
            as <- rpar (force (primesBtwn 2 hf))
            bs <- rpar (force (primesBtwn (hf + 1) lim))
            return (as ++ bs)

      forkIO $ putStrLn ("\nPrimes are: " ++ (show result) ++ " for " ++ n ++ "\n")
      loop
```

```
-- Compute primes between two integers
-- Deliberately inefficient for demonstration purposes
primesBtwn n m = eratos [n..m]
  where
    eratos []     = []
    eratos (p:xs) = p : eratos (xs `minus` [p, p+p..])
```

Ejecutando esto demostrará el comportamiento concurrente:

```
Enter a number
12
Enter a number

Primes are: [2,3,5,7,8,9,10,11,12] for 12

100
Enter a number

Primes are:
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,7
for 100

200000000
Enter a number
-- waiting for 200000000
200
Enter a number

Primes are:
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,102,103,104,105,106,107,108
for 200

-- still waiting for 200000000
```

rseq

Podemos usar `rseq :: Strategy a` para forzar un argumento a una forma normal de cabeza débil:

```
f1 :: [Int]
f1 = [1..100000000]

f2 :: [Int]
f2 = [1..200000000]

main = runEval $ do
  a <- rpar (f1) -- this'll take a while...
  b <- rpar (f2) -- this'll take a while and then some...
  rseq a
  return (a,b)
```

Esto cambia sutilmente la semántica del ejemplo `rpar` ; mientras que el último sería volver de inmediato, mientras que el cálculo de los valores en el fondo, este ejemplo esperará hasta que `a` se puede evaluar a WHNF.

Lea Paralelismo en línea: https://riptutorial.com/es/haskell/topic/6887/paralelismo

Capítulo 54: Plantilla Haskell & QuasiQuotes

Observaciones

¿Qué es la plantilla Haskell?

La plantilla Haskell se refiere a las instalaciones de meta-programación de plantillas incorporadas en GHC Haskell. El documento que describe la implementación original se puede encontrar aquí .

¿Qué son las etapas? (O, ¿cuál es la restricción de la etapa?)

Las etapas se refieren a *cuando* se ejecuta el código. Normalmente, el código se ejerce solo en tiempo de ejecución, pero con Template Haskell, el código puede ejecutarse en tiempo de compilación. El código "normal" es la etapa 0 y el código de tiempo de compilación es la etapa 1.

La restricción de etapa se refiere al hecho de que un programa de etapa 0 no puede ejecutarse en la etapa 1; esto sería equivalente a poder ejecutar cualquier programa *regular* (no solo un meta-programa) en el momento de la compilación.

Por convención (y para simplificar la implementación), el código en el módulo actual siempre es la etapa 0 y el código importado de todos los demás módulos es la etapa 1. Por esta razón, solo las expresiones de otros módulos pueden empalmarse.

Tenga en cuenta que un programa de etapa 1 es una expresión de etapa 0 de tipo `Q Exp` , `Q Type` , etc .; pero lo contrario no es cierto: no todos los valores (programa de la etapa 0) de tipo `Q Exp` son programas de la etapa 1,

Además, como los empalmes se pueden anidar, los identificadores pueden tener etapas superiores a 1. La restricción de etapa puede generalizarse: un programa de etapa *n* no puede ejecutarse en ninguna etapa *m> n* . Por ejemplo, uno puede ver referencias a tales etapas mayores que 1 en ciertos mensajes de error:

```
>:t [| \x -> $x |]

<interactive>:1:10: error:
    * Stage error: `x' is bound at stage 2 but used at stage 1
    * In the untyped splice: $x
      In the Template Haskell quotation [| \ x -> $x |]
```

¿El uso de Template Haskell causa errores

no relacionados con el alcance de identificadores no relacionados?

Normalmente, todas las declaraciones en un solo módulo de Haskell se pueden considerar como todas recursivas mutuamente. En otras palabras, cada declaración de nivel superior está dentro del alcance de todas las demás en un solo módulo. Cuando Template Haskell está habilitado, las reglas de alcance cambian: el módulo se divide en grupos de código separados por empalmes TH, y cada grupo es recursivo mutuamente, y cada grupo está dentro del alcance de todos los grupos adicionales.

Examples

El tipo q

El constructor de tipo `Q :: * -> *` definido en `Language.Haskell.TH.Syntax` es un tipo abstracto que representa los cálculos que tienen acceso al entorno de tiempo de compilación del módulo en el que se ejecuta el cálculo. El tipo `Q` también maneja la sustitución de variables, llamada *captura de nombre* por TH (y se explica aquí). Todos los empalmes tienen el tipo `QX` para algunas `X`

El entorno de tiempo de compilación incluye:

- identificadores en el ámbito e información sobre dichos identificadores,
    - tipos de funciones
    - tipos y fuentes de datos tipos de constructores
    - Especificación completa de las declaraciones de tipo (clases, familias de tipo)
- la ubicación en el código fuente (línea, columna, módulo, paquete) donde se produce el empalme
- Fijidades de funciones (GHC 7.10)
- extensiones GHC habilitadas (GHC 8.0)

El tipo `Q` también tiene la capacidad de generar nombres nuevos, con la función `newName :: String -> Q Name` . Tenga en cuenta que el nombre no está vinculado en ningún lugar de forma implícita, por lo que el usuario debe vincularlo por sí mismo, por lo que asegurarse de que el uso resultante del nombre esté bien incluido es responsabilidad del usuario.

`Q` tiene instancias para `Functor,Monad,Applicative` y esta es la interfaz principal para manipular los valores de `Q` , junto con los combinadores proporcionados en `Language.Haskell.TH.Lib` , que definen una función auxiliar para cada constructor de la forma TH:

```
LitE :: Lit -> Exp
litE :: Lit -> ExpQ

AppE :: Exp -> Exp -> Exp
appE :: ExpQ -> ExpQ -> ExpQ
```

Tenga en cuenta que `ExpQ` , `TypeQ` , `DecsQ` y `PatQ` son sinónimos de los tipos de AST que

normalmente se almacenan dentro del tipo `Q`

La biblioteca TH proporciona una función `runQ :: Quasi m => Q a -> ma` , y hay una instancia de `Quasi IO` , por lo que parece que el tipo `Q` es solo una `IO` elegante. Sin embargo, el uso de `runQ :: Q a -> IO a` produce una acción de `IO` que *no* tiene acceso a ningún entorno de tiempo de compilación; solo está disponible en el tipo de `Q` real. Tales acciones de `IO` fallarán en el tiempo de ejecución si se intenta acceder a dicho entorno.

Un curry n-arity

Lo familiar

```
curry :: ((a,b) -> c) -> a -> b -> c
curry = \f a b -> f (a,b)
```

La función puede generalizarse a tuplas de aridad arbitraria, por ejemplo:

```
curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
curry4 :: ((a, b, c, d) -> e) -> a -> b -> c -> d -> e
```

Sin embargo, escribir tales funciones para tuplas de aridad 2 a (p. Ej., 20 a mano) sería tedioso (e ignorar el hecho de que la presencia de 20 tuplas en su programa casi con toda seguridad indica problemas de diseño que deberían solucionarse con los registros).

Podemos usar Template Haskell para producir tales funciones `curryN` para `n` arbitrario:

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Monad (replicateM)
import Language.Haskell.TH (ExpQ, newName, Exp(..), Pat(..))
import Numeric.Natural (Natural)

curryN :: Natural -> Q Exp
```

La función `curryN` toma un número natural y produce la función curry de esa aridad, como un AST de Haskell.

```
curryN n = do
  f  <- newName "f"
  xs <- replicateM (fromIntegral n) (newName "x")
```

Primero producimos *nuevas* variables de tipo para cada uno de los argumentos de la función, uno para la función de entrada y uno para cada uno de los argumentos de dicha función.

```
  let args = map VarP (f:xs)
```

La expresión `args` representa el patrón `f x1 x2 .. xn` . Tenga en cuenta que un patrón es una entidad sintáctica separada: podríamos tomar este mismo patrón y colocarlo en un lambda, o una función de enlace, o incluso el LHS de un enlace de dejar (que sería un error).

```
      ntup = TupE (map VarE xs)
```

La función debe construir la tupla de argumentos a partir de la secuencia de argumentos, que es lo que hemos hecho aquí. Observe la distinción entre variables de patrón (`VarP`) y variables de expresión (`VarE`).

```
  return $ LamE args (AppE (VarE f) ntup)
```

Finalmente, el valor que producimos es el AST `\f x1 x2 .. xn -> f (x1, x2, .. , xn)` .

También podríamos haber escrito esta función usando citas y constructores 'levantados':

```
...
import Language.Haskell.TH.Lib

curryN' :: Natural -> ExpQ
curryN' n = do
  f  <- newName "f"
  xs <- replicateM (fromIntegral n) (newName "x")
  lamE (map varP (f:xs))
       [| $(varE f) $(tupE (map varE xs)) |]
```

Tenga en cuenta que las citas deben ser sintácticamente válidas, por lo que `[| \ $(map varP (f:xs)) -> .. |]` no es válido, porque en Haskell normal no hay forma de declarar una 'lista' de patrones; lo anterior se interpreta como `\ var -> ..` y se espera que la expresión empalmada tenga el tipo `PatQ` , es decir, un solo patrón, no una lista de patrones.

Finalmente, podemos cargar esta función TH en GHCi:

```
>:set -XTemplateHaskell
>:t $(curryN 5)
$(curryN 5)
  :: ((t1, t2, t3, t4, t5) -> t) -> t1 -> t2 -> t3 -> t4 -> t5 -> t
>$(curryN 5) (\(a,b,c,d,e) -> a+b+c+d+e) 1 2 3 4 5
15
```

Este ejemplo está adaptado principalmente desde aquí .

Sintaxis de plantilla Haskell y cuasiquotes

La plantilla Haskell está habilitada por la extensión `-XTemplateHaskell` GHC. Esta extensión habilita todas las características sintácticas más detalladas en esta sección. Los detalles completos sobre Template Haskell están dados por la guía del usuario .

Empalmes

- Un empalme es una nueva entidad sintáctica habilitada por Template Haskell, escrita como `$(...)` , donde `(...)` es una expresión.

- No debe haber un espacio entre `$` y el primer carácter de la expresión; y Template Haskell anula el análisis del operador `$` , por ejemplo, `f$g` normalmente se analiza como `($) fg` mientras que con la plantilla Haskell habilitada, se analiza como un empalme.
- Cuando aparece un empalme en el nivel superior, se puede omitir el `$` . En este caso, la expresión empalmada es la línea completa.
- Un empalme representa el código que se ejecuta en tiempo de compilación para producir un AST de Haskell, y ese AST se compila como un código de Haskell y se inserta en el programa
- Los empalmes pueden aparecer en lugar de: expresiones, patrones, tipos y declaraciones de nivel superior. El tipo de expresión empalmada, en cada caso respectivamente, es `Q Exp` , `Q Pat` , `Q Type` , `Q [Decl]` . Tenga en cuenta que los empalmes de declaración *solo* pueden aparecer en el nivel superior, mientras que los otros pueden estar dentro de otras expresiones, patrones o tipos, respectivamente.

Citas de expresión (nota: *no* una cotización)

- Una cita de expresión es una nueva entidad sintáctica escrita como una de:
  - `[e|..|]` o `[||..||]` - `..` es una expresión y la cita tiene el tipo `Q Exp` ;
  - `[p|..|]` - `..` es un patrón y la cita tiene el tipo `Q Pat` ;
  - `[t|..|]` - `..` es un tipo y la cita tiene el tipo `Q Type` ;
  - `[d|..|]` - `..` es una lista de declaraciones y la cita tiene el tipo `Q [Dec]` .
- Una cita de expresión toma un programa de tiempo de compilación y produce el AST representado por ese programa.
- El uso de un valor en una cita (por ejemplo, `\x -> [| x |]`) sin un empalme corresponde al azúcar sintáctico para `\x -> [| $(lift x) |]` , donde `lift :: Lift t => t -> Q Exp` proviene de la clase

```
class Lift t where
  lift :: t -> Q Exp
  default lift :: Data t => t -> Q Exp
```

Empalmes mecanografiados y citas

- Los empalmes escritos son similares a los empalmes mencionados anteriormente (sin tipo), y se escriben como `$$(..)` donde `(..)` es una expresión.
- Si `e` tiene el tipo `Q (TExp a)` entonces `$$e` tiene el tipo `a` .
- Las citas escritas toman la forma `[||..||]` donde `..` es una expresión de tipo `a` ; la cita resultante tiene el tipo `Q (TExp a)` .

- La expresión escrita se puede convertir a una sin tipo: `unType :: TExp a -> Exp` .

Cuasi Citas

- QuasiQuotes generaliza las citas de expresión: anteriormente, el analizador utilizado por la expresión de la cita es uno de un conjunto fijo (`e,p,t,d`), pero QuasiQuotes permite definir un analizador personalizado y utilizarlo para producir código en el momento de la compilación. Las casi comillas pueden aparecer en todos los mismos contextos que las citas regulares.
- Una casi cita se escribe como `[iden|...|]` , donde `iden` es un identificador de tipo `Language.Haskell.TH.Quote.QuasiQuoter` .
- Un `QuasiQuoter` está compuesto simplemente por cuatro analizadores, uno para cada uno de los diferentes contextos en los que pueden aparecer las citas:

```
data QuasiQuoter = QuasiQuoter { quoteExp  :: String -> Q Exp,
                                 quotePat  :: String -> Q Pat,
                                 quoteType :: String -> Q Type,
                                 quoteDec  :: String -> Q [Dec] }
```

Los nombres

- Los identificadores de Haskell están representados por el tipo `Language.Haskell.TH.Syntax.Name` . Los nombres forman las hojas de sintaxis abstracta que representan los programas de Haskell en la Plantilla Haskell.
- Un identificador que se encuentra actualmente dentro del alcance puede convertirse en un nombre con: `'e` o `'T` En el primer caso, `e` se interpreta en el ámbito de expresión, mientras que en el segundo caso, `T` está en el ámbito de tipo (recordando que los constructores de tipos y valores pueden compartir el nombre sin ambigüedad en Haskell).

Lea Plantilla Haskell & QuasiQuotes en línea: https://riptutorial.com/es/haskell/topic/5216/plantilla-haskell--amp--quasiquotes

Capítulo 55: Plátano reactivo

Examples

Inyectando eventos externos en la biblioteca.

Este ejemplo no está vinculado a ningún conjunto de herramientas GUI concreto, como hace el caso de reactivo-banana-wx, por ejemplo. En su lugar, muestra cómo inyectar acciones arbitrarias de `IO` / `IO` en la maquinaria de FRP.

El módulo `Control.Event.Handler` proporciona una función `addHandler` que crea un par de `AddHandler a` y `a -> IO ()` . El primero es utilizado por el propio banana reactiva para obtener un `Event a` valor, mientras que el segundo es una función simple que se usa para desencadenar el evento correspondiente.

```
import Data.Char (toUpper)

import Control.Event.Handler
import Reactive.Banana

main = do
    (inputHandler, inputFire) <- newAddHandler
```

En nuestro caso el `a` parámetro del controlador es de tipo `String` , pero el código que permite inferir que el compilador se escribirá más tarde.

Ahora definimos la `EventNetwork` que describe nuestro sistema controlado por FRP. Esto se hace usando la función de `compile` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        inputEvent <- fromAddHandler inputHandler
```

La función `fromAddHandler` transforma `AddHandler a` valor `AddHandler a` en un `Event a` , que se trata en el siguiente ejemplo.

Finalmente, lanzamos nuestro "bucle de eventos", que dispararía eventos en la entrada del usuario:

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
    forever $ do
        input <- getLine
        inputFire input
```

Tipo de evento

En reactive-banana, el tipo de `Event` representa una secuencia de algunos eventos en el tiempo. Un `Event` es similar a una señal de impulso analógica en el sentido de que no es continua en el tiempo. Como resultado, `Event` es una instancia de la clase de tipos de `Functor` solamente. No puedes combinar dos `Event` s juntos porque pueden disparar en diferentes momentos. Puede hacer algo con el valor [actual] de un `Event` y reaccionar a él con alguna acción `IO` .

Las transformaciones en el valor del `Event` se realizan usando `fmap` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        inputEvent <- fromAddHandler inputHandler
        -- turn all characters in the signal to upper case
        let inputEvent' = fmap (map toUpper) inputEvent
```

Reaccionar ante un `Event` se hace de la misma manera. Primero `fmap` con una acción de tipo `a -> IO ()` y luego `reactimate` para `reactimate` función:

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        inputEvent <- fromAddHandler inputHandler
        -- turn all characters in the signal to upper case
        let inputEvent' = fmap (map toUpper) inputEvent
        let inputEventReaction = fmap putStrLn inputEvent' -- this has type `Event (IO ())
        reactimate inputEventReaction
```

Ahora, cada vez que se `inputFire "something"` , se imprimirá `"SOMETHING"` .

Tipo de comportamiento

Para representar señales continuas, características reactivas de banana `Behavior a` tipo. A diferencia del `Event` , un `Behavior` es un `Applicative` , que le permite combinar n `Behavior` usando una función pura n-aría (usando `<$>` y `<*>`).

Para obtener un `Behavior a` del `Event a` hay `Event a` función de `accumE` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
```

`accumE` toma el valor inicial de `Behavior` y un `Event` , que contiene una función que lo establecería en el nuevo valor.

Al igual que con los `Event` s, puede usar `fmap` para trabajar con el valor del `Behavior` actual, pero también puede combinarlos con `(<*>)` .

```
main = do
    (inputHandler, inputFire) <- newAddHandler
```

```
    compile $ do
        ...
        inputBehavior  <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        inputBehavior' <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        let constantTrueBehavior = (==) <$> inputBehavior <*> inputBehavior'
```

Para reaccionar ante los cambios de `Behavior` hay una función de `changes` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior  <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        inputBehavior' <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        let constantTrueBehavior = (==) <$> inputBehavior <*> inputBehavior'
        inputChanged <- changes inputBehavior
```

Lo único que se debe tener en cuenta es que los `changes` devuelven el `Event (Future a)` lugar del `Event a` . Debido a esto, `reactimate'` se debe utilizar en lugar de `reactimate` . La razón detrás de esto se puede obtener de la documentación.

Actuating EventNetworks

`EventNetwork` devueltos por `compile` deben activarse antes de que los eventos reactivados tengan un efecto.

```
main = do
    (inputHandler, inputFire) <- newAddHandler

    eventNetwork <- compile $ do
        inputEvent <- fromAddHandler inputHandler
        let inputEventReaction = fmap putStrLn inputEvent
        reactimate inputEventReaction

    inputFire "This will NOT be printed to the console!"
    actuate eventNetwork
    inputFire "This WILL be printed to the console!"
```

Lea Plátano reactivo en línea: https://riptutorial.com/es/haskell/topic/4186/platano-reactivo

Capítulo 56: Plegable

Introducción

`Foldable` es la clase de tipos `t :: * -> *` que admite una operación de *plegado* . Un pliegue agrega los elementos de una estructura en un orden bien definido, utilizando una función de combinación.

Observaciones

Si `t` es `Foldable` significa que para cualquier valor `ta` sabemos cómo acceder a todos los elementos de `a` desde "adentro" de `ta` en un orden lineal fijo. Este es el significado de `foldMap :: Monoid m => (a -> m) -> (ta -> m)` : "visitamos" cada elemento con una función de resumen y rompemos todos los resúmenes juntos. El orden de respeto de los `Monoid` (pero son invariantes a diferentes agrupaciones).

Examples

Contando los elementos de una estructura plegable.

`length` cuenta las ocurrencias de los elementos `a` en una estructura plegable `ta` .

```
ghci> length [7, 2, 9]  -- t ~ []
3
ghci> length (Right 'a')  -- t ~ Either e
1  -- 'Either e a' may contain zero or one 'a'
ghci> length (Left "foo")  -- t ~ Either String
0
ghci> length (3, True)  -- t ~ (,) Int
1  -- '(c, a)' always contains exactly one 'a'
```

`length` se define como equivalente a:

```
class Foldable t where
    -- ...
    length :: t a -> Int
    length = foldl' (\c _ -> c+1) 0
```

Tenga en cuenta que este tipo de retorno `Int` restringe las operaciones que pueden realizarse en valores obtenidos por llamadas a la función de `length` . `fromIntegral` es una función útil que nos permite lidiar con este problema.

Doblando una estructura al revés

Cualquier pliegue se puede ejecutar en la dirección opuesta con la ayuda del `Dual` monoide , que voltea un monoide existente para que la agregación se desplace hacia atrás.

```
newtype Dual a = Dual { getDual :: a }

instance Monoid m => Monoid (Dual m) where
    mempty = Dual mempty
    (Dual x) `mappend` (Dual y) = Dual (y `mappend` x)
```

Cuando el monoide subyacente de una llamada `foldMap` se invierte con `Dual` , el pliegue se ejecuta hacia atrás; el siguiente tipo `Reverse` se define en `Data.Functor.Reverse` :

```
newtype Reverse t a = Reverse { getReverse :: t a }

instance Foldable t => Foldable (Reverse t) where
    foldMap f = getDual . foldMap (Dual . f) . getReverse
```

Podemos usar esta maquinaria para escribir un `reverse` terso para las listas:

```
reverse :: [a] -> [a]
reverse = toList . Reverse
```

Una instancia de Plegable para un árbol binario

Para crear una instancia de `Foldable` , debe proporcionar una definición de al menos `foldMap` o `foldr` .

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

instance Foldable Tree where
    foldMap f Leaf = mempty
    foldMap f (Node l x r) = foldMap f l `mappend` f x `mappend` foldMap f r

    foldr f acc Leaf = acc
    foldr f acc (Node l x r) = foldr f (f x (foldr f acc r)) l
```

Esta implementación realiza un recorrido en orden del árbol.

```
ghci> let myTree = Node (Node Leaf 'a' Leaf) 'b' (Node Leaf 'c' Leaf)

--    +--'b'--+
--    |       |
-- +-'a'-+ +-'c'-+
-- |     | |     |
-- *     * *     *

ghci> toList myTree
"abc"
```

La extensión `DeriveFoldable` permite a GHC generar instancias `Foldable` basadas en la estructura del tipo. Podemos variar el orden del recorrido escrito por la máquina ajustando el diseño del constructor `Node` .

```
data Inorder a = ILeaf
```

```
               | INode (Inorder a) a (Inorder a)  -- as before
               deriving Foldable

data Preorder a = PrLeaf
                | PrNode a (Preorder a) (Preorder a)
                deriving Foldable

data Postorder a = PoLeaf
                 | PoNode (Postorder a) (Postorder a) a
                 deriving Foldable

-- injections from the earlier Tree type
inorder :: Tree a -> Inorder a
inorder Leaf = ILeaf
inorder (Node l x r) = INode (inorder l) x (inorder r)

preorder :: Tree a -> Preorder a
preorder Leaf = PrLeaf
preorder (Node l x r) = PrNode x (preorder l) (preorder r)

postorder :: Tree a -> Postorder a
postorder Leaf = PoLeaf
postorder (Node l x r) = PoNode (postorder l) (postorder r) x

ghci> toList (inorder myTree)
"abc"
ghci> toList (preorder myTree)
"bac"
ghci> toList (postorder myTree)
"acb"
```

Aplanando una estructura plegable en una lista

`toList` aplana una estructura `Foldable t a` en una lista de `a` s.

```
ghci> toList [7, 2, 9]  -- t ~ []
[7, 2, 9]
ghci> toList (Right 'a')  -- t ~ Either e
"a"
ghci> toList (Left "foo")  -- t ~ Either String
[]
ghci> toList (3, True)  -- t ~ (,) Int
[True]
```

`toList` se define como equivalente a:

```
class Foldable t where
    -- ...
    toList :: t a -> [a]
    toList = foldr (:) []
```

Realización de un efecto secundario para cada elemento de una estructura plegable

`traverse_` ejecuta una acción `Applicative` para cada elemento en una estructura `Foldable` . Ignora

el resultado de la acción, manteniendo solo los efectos secundarios. (Para una versión que no descarta resultados, use `Traversable`).

```
-- using the Writer applicative functor (and the Sum monoid)
ghci> runWriter $ traverse_ (\x -> tell (Sum x)) [1,2,3]
((),Sum {getSum = 6})
-- using the IO applicative functor
ghci> traverse_ putStrLn (Right "traversing")
traversing
ghci> traverse_ putStrLn (Left False)
-- nothing printed
```

`for_` es `traverse_` con los argumentos volteados. Se asemeja a un bucle `foreach` en un lenguaje imperativo.

```
ghci> let greetings = ["Hello", "Bonjour", "Hola"]
ghci> :{
ghci|     for_ greetings $ \greeting -> do
ghci|         print (greeting ++ " Stack Overflow!")
ghci| :}
"Hello Stack Overflow!"
"Bonjour Stack Overflow!"
"Hola Stack Overflow!"
```

`sequenceA_` colapsa un `Foldable` lleno de acciones de `Applicative` en una sola acción, ignorando el resultado.

```
ghci> let actions = [putStrLn "one", putStLn "two"]
ghci> sequenceA_ actions
one
two
```

`traverse_` se define como equivalente a:

```
traverse_ :: (Foldable t, Applicative f) => (a -> f b) -> t a -> f ()
traverse_ f = foldr (\x action -> f x *> action) (pure ())
```

`sequenceA_` se define como:

```
sequenceA_ :: (Foldable t, Applicative f) -> t (f a) -> f ()
sequenceA_ = traverse_ id
```

Además, cuando el `Foldable` también es un `Functor` , `traverse_` y `sequenceA_` tienen la siguiente relación:

```
traverse_ f = sequenceA_ . fmap f
```

Aplanando una estructura plegable en un monoide

`foldMap` asigna cada elemento de la estructura plegable a un `Monoid` , y luego los combina en un solo valor.

`foldMap` y `foldr` se pueden definir en términos de otros, lo que significa que las instancias de `Foldable` solo necesitan dar una definición para uno de ellos.

```
class Foldable t where
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty
```

Ejemplo de uso con el `Product` monoide :

```
product :: (Num n, Foldable t) => t n -> n
product = getProduct . foldMap Product
```

Definición de plegable

```
class Foldable t where
    {-# MINIMAL foldMap | foldr #-}

    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty

    foldr :: (a -> b -> b) -> b -> t a -> b
    foldr f z t = appEndo (foldMap (Endo #. f) t) z

    -- and a number of optional methods
```

Intuitivamente (aunque no técnicamente), `Foldable` estructuras son contenedores de elementos de `a` que permiten el acceso a sus elementos en un orden bien definido. La operación `foldMap` asigna cada elemento del contenedor a un `Monoid` y los colapsa utilizando la estructura `Monoid` .

Comprobando si una estructura plegable está vacía

`null` devuelve `True` si no hay elementos `a` en una estructura plegable `ta` , y `False` si hay uno o más. Las estructuras para las cuales `null` es `True` tienen una `length` de 0.

```
ghci> null []
True
ghci> null [14, 29]
False
ghci> null Nothing
True
ghci> null (Right 'a')
False
ghci> null ('x', 3)
False
```

`null` se define como equivalente a:

```
class Foldable t where
    -- ...
    null :: t a -> Bool
    null = foldr (\_ _ -> False) True
```

Lea Plegable en línea: https://riptutorial.com/es/haskell/topic/753/plegable

Capítulo 57: Polimorfismo de rango arbitrario con RankNTypes

Introducción

El sistema de tipos de GHC admite la cuantificación universal explícita de rango arbitrario en tipos mediante el uso de las extensiones de lenguaje `Rank2Types` y `RankNTypes` .

Sintaxis

- La cuantificación arbitraria de rango se habilita con la extensión de lenguaje `Rank2Types` o `RankNTypes` .
- Con esta extensión habilitada, la palabra clave `forall` se puede usar para agregar una cuantificación de rango superior.

Examples

RankNTypes

StackOverflow me obliga a tener un ejemplo. Si se aprueba este tema, deberíamos mover este ejemplo aquí.

Lea Polimorfismo de rango arbitrario con RankNTypes en línea:
https://riptutorial.com/es/haskell/topic/8984/polimorfismo-de-rango-arbitrario-con-rankntypes

Capítulo 58: Probando con Tasty

Examples

SmallCheck, QuickCheck y HUnit

```
import Test.Tasty
import Test.Tasty.SmallCheck as SC
import Test.Tasty.QuickCheck as QC
import Test.Tasty.HUnit


main :: IO ()
main = defaultMain tests

tests :: TestTree
tests = testGroup "Tests" [smallCheckTests, quickCheckTests, unitTests]

smallCheckTests :: TestTree
smallCheckTests = testGroup "SmallCheck Tests"
  [ SC.testProperty "String length <= 3" $
      \s -> length (take 3 (s :: String)) <= 3
  , SC.testProperty "String length <= 2" $  -- should fail
      \s -> length (take 3 (s :: String)) <= 2
  ]

quickCheckTests :: TestTree
quickCheckTests = testGroup "QuickCheck Tests"
  [ QC.testProperty "String length <= 5" $
      \s -> length (take 5 (s :: String)) <= 5
  , QC.testProperty "String length <= 4" $  -- should fail
      \s -> length (take 5 (s :: String)) <= 4
  ]

unitTests :: TestTree
unitTests = testGroup "Unit Tests"
  [ testCase "String comparison 1" $
      assertEqual "description" "OK" "OK"

  , testCase "String comparison 2" $  -- should fail
      assertEqual "description" "fail" "fail!"
  ]
```

Instalar paquetes:

```
cabal install tasty-smallcheck tasty-quickcheck tasty-hunit
```

Ejecutar con cabal

```
cabal exec runhaskell test.hs
```

Lea Probando con Tasty en línea: https://riptutorial.com/es/haskell/topic/3816/probando-con-tasty

Capítulo 59: Profesor

Introducción

`Profunctor` es una clase de tipos proporcionada por el `profunctors` paquete en `Data.Profunctor` .

Consulte la sección "Comentarios" para obtener una explicación completa.

Sintaxis

- dimap :: Profunctor p => (a -> b) -> (c -> d) -> pbc -> pad
- lmap :: Profunctor p => (a -> b) -> pbc -> pac
- rmap :: Profunctor p => (b -> c) -> pab -> pac
- ID de dimap id = id
- lmap id = id
- rmap id = id
- dimap fg = lmap f. rmap g
- lmap f = dimap f id
- rmap f = dimap id f

Observaciones

Los expertos son, según lo descrito por los documentos en Hackage, "un bifunctor donde el primer argumento es contravariante y el segundo argumento es covariante".

Entonces, ¿qué significa esto? Bueno, un bifunctor es como un funtor normal, excepto que tiene dos parámetros en lugar de uno, cada uno con su propia función `fmap` para mapear en él.

Ser "covariante" significa que el segundo argumento para un profunctor es como un funtor normal: su función de mapeo (`rmap`) tiene una firma de tipo de `Profunctor p => (b -> c) -> pab -> pac` . Simplemente mapea la función en el segundo argumento.

Ser "contravariante" hace que el primer argumento sea un poco más extraño. En lugar de mapear como un funtor normal, su función de mapeo (`lmap`) tiene una firma de tipo de `Profunctor p => (a -> b) -> pbc -> pac` . Esta asignación aparentemente hacia atrás tiene más sentido para las entradas a una función: ejecutaría `a -> b` en la entrada, y luego la otra función, dejando la nueva entrada como `a` .

Nota: la denominación de los funtores normales de un argumento es un poco engañosa: la clase de tipos `Functor` implementa funtores "covariantes", mientras que los functores "contravariantes" se implementan en la clase de `Data.Functor.Contravariant` `Contravariant` en `Data.Functor.Contravariant` , y anteriormente el (nombre erróneo) `Cofunctor` typeclass en `Data.Cofunctor` .

Examples

(->) Profesor

(->) es un ejemplo simple de un profunctor: el argumento de la izquierda es la entrada a una función, y el argumento de la derecha es el mismo que el de la instancia del functor del lector.

```
instance Profunctor (->) where
    lmap f g = g . f
    rmap f g = g . g
```

Lea Profesor en línea: https://riptutorial.com/es/haskell/topic/9694/profesor

Capítulo 60: Proxies

Examples

Usando Proxy

El tipo `Proxy :: k -> *` , que se encuentra en `Data.Proxy` , se usa cuando se necesita dar cierta información de tipo al compilador, por ejemplo, para elegir una instancia de clase de tipo, que no obstante es irrelevante en el tiempo de ejecución.

```
{-# LANGUAGE PolyKinds #-}

data Proxy a = Proxy
```

Las funciones que usan un `Proxy` generalmente usan `ScopedTypeVariables` para elegir una instancia de clase de tipo basada en `a` tipo.

Por ejemplo, el ejemplo clásico de una función ambigua,

```
showread :: String -> String
showread = show . read
```

lo que resulta en un error de tipo porque el elaborador no sabe qué instancia de `Show` o `Read` usar, puede resolverse usando un `Proxy` :

```
{-# LANGUAGE ScopedTypeVariables #-}

import Data.Proxy

showread :: forall a. (Show a, Read a) => Proxy a -> String -> String
showread _ = (show :: a -> String) . read
```

Cuando se llama a una función con `Proxy` , es necesario utilizar una anotación de tipo para declarar la cual `a` que quería decir.

```
ghci> showread (Proxy :: Proxy Int) "3"
"3"
ghci> showread (Proxy :: Proxy Bool) "'m'"  -- attempt to parse a char literal as a Bool
"*** Exception: Prelude.read: no parse
```

El lenguaje "proxy polimórfico"

Dado que el `Proxy` no contiene información de tiempo de ejecución, nunca es necesario realizar una coincidencia de patrones en el constructor `Proxy` . Por lo tanto, un lenguaje común es abstraer sobre el tipo de datos `Proxy` usando una variable de tipo.

```
showread :: forall proxy a. (Show a, Read a) => proxy a -> String -> String
```

```
showread _ = (show :: a -> String) . read
```

Ahora, si tiene un `fa` en el alcance de alguna `f` , no necesita escribir `Proxy :: Proxy a` cuando llame a `f` .

```
ghci> let chars = "foo"  -- chars :: [Char]
ghci> showread chars "'a'"
"'a'"
```

Proxy es como ()

Dado que `Proxy` no contiene información de tiempo de ejecución, siempre puede escribir una transformación natural `fa -> Proxy a` para cualquier `f` .

```
proxy :: f a -> Proxy a
proxy _ = Proxy
```

Esto es así como cualquier valor dado siempre puede borrarse a `()` :

```
unit :: a -> ()
unit _ = ()
```

Técnicamente, `Proxy` es el objeto terminal en la categoría de los funtores, al igual que `()` es el objeto terminal en la categoría de valores.

Lea Proxies en línea: https://riptutorial.com/es/haskell/topic/8025/proxies

Capítulo 61: Reglas de reescritura (GHC)

Examples

Usando reglas de reescritura en funciones sobrecargadas

En esta pregunta , @Viclib preguntó sobre el uso de reglas de reescritura para explotar las leyes de clase de tipos y eliminar algunas llamadas a funciones sobrecargadas:

> Cuidado con la siguiente clase:
>
> ```
> class ListIsomorphic l where
>     toList   :: l a -> [a]
>     fromList  :: [a] -> l a
> ```
>
> También exijo que `toList . fromList == id`. ¿Cómo escribo las reglas de reescritura para decirle a GHC que haga esa sustitución?

Este es un caso de uso un tanto complicado para el mecanismo de reglas de reescritura de GHC, porque las funciones sobrecargadas se reescriben en sus métodos de instancia específicos mediante reglas que están creadas implícitamente detrás de escena por GHC (por lo tanto, algo como `fromList :: Seq a -> [a]` sería reescrito en `Seq$fromList` etc.).

Sin embargo, al reescribir primero a `toList` y `fromList` a métodos no tipográficos no tipográficos, podemos protegerlos de una reescritura prematura y preservarlos hasta que la regla para la composición pueda activarse:

```
{-# RULES
  "protect toList"   toList = toList';
  "protect fromList" fromList = fromList';
  "fromList/toList"  forall x . fromList' (toList' x) = x; #-}

{-# NOINLINE [0] fromList' #-}
fromList' :: (ListIsomorphic l) => [a] -> l a
fromList' = fromList

{-# NOINLINE [0] toList' #-}
toList' :: (ListIsomorphic l) => l a -> [a]
toList' = toList
```

Lea Reglas de reescritura (GHC) en línea: https://riptutorial.com/es/haskell/topic/4914/reglas-de-reescritura--ghc-

Capítulo 62: Rigor

Examples

Patrones de explosión

Los patrones anotados con una explosión (`!`) Se evalúan estrictamente en lugar de perezosamente.

```
foo (!x, y) !z = [x, y, z]
```

En este ejemplo, tanto `x` como `z` se evaluarán con una forma normal de encabezado débil antes de devolver la lista. Es equivalente a:

```
foo (x, y) z = x `seq` z `seq` [x, y, z]
```

Los patrones de Bang se habilitan utilizando la extensión de lenguaje `BangPatterns` Haskell 2010.

Formas normales

Este ejemplo proporciona una breve descripción general: para ver una explicación más detallada de *formas* y ejemplos *normales* , consulte esta pregunta .

Forma normal reducida

La forma normal reducida (o simplemente la forma normal, cuando el contexto es claro) de una expresión es el resultado de evaluar todas las subexpresiones reducibles en la expresión dada. Debido a la semántica no estricta de Haskell (normalmente llamada *pereza*), una subexpresión no es reducible si está bajo un aglutinante (es decir, una abstracción lambda - `\x -> ..`). La forma normal de una expresión tiene la propiedad de que, si existe, es única.

En otras palabras, no importa (en términos de semántica denotacional) en qué orden se reducen las subexpresiones. Sin embargo, la clave para escribir los programas de Haskell de rendimiento a menudo es garantizar que la expresión correcta se evalúe en el momento adecuado, es decir, la comprensión de la semántica operacional.

Una expresión cuya forma normal es en sí misma se dice que está *en forma normal* .

Algunas expresiones, por ejemplo, `let x = 1:x in x` , no tienen forma normal, pero siguen siendo productivas. La expresión de ejemplo todavía tiene un *valor* , si se admiten valores infinitos, que aquí está la lista `[1,1, ...]` . Otras expresiones, como `let y = 1+y in y` , no tienen ningún valor o su valor `undefined` está `undefined` .

Forma normal de la cabeza débil

El RNF corresponde a la evaluación completa de una expresión. Del mismo modo, la forma normal de la cabeza débil (WHNF, por sus siglas en inglés) corresponde a la evaluación del *encabezado* de la expresión. El encabezado de una expresión `e` se evalúa completamente si `e` es una aplicación `Con e1 e2 .. en` y `Con` es un constructor; o una abstracción `\x -> e1` ; o una aplicación parcial `f e1 e2 .. en` , donde aplicación parcial significa `f` toma más de `n` argumentos (o, de manera equivalente, el tipo de `e` es un tipo de función). En cualquier caso, las subexpresiones `e1..en` pueden evaluarse o no evaluarse para que la expresión esté en WHNF, incluso pueden estar `undefined` .

La semántica de evaluación de Haskell se puede describir en términos de WHNF: para evaluar una expresión `e` , primero evalúela a WHNF, luego evalúe recursivamente todas sus subexpresiones de izquierda a derecha.

La primitiva `seq` función se utiliza para evaluar una expresión a WHNF. `seq xy` es denotacionalmente igual a `y` (el valor de `seq xy` es precisamente `y`); además, `x` se evalúa como WHNF cuando `y` se evalúa como WHNF. Una expresión también se puede evaluar a WHNF con un patrón de bang (habilitado por la extensión `-XBangPatterns`), cuya sintaxis es la siguiente:

```
f !x y = ...
```

En que `x` se evaluará a WHNF cuando `f` se evalúa, mientras que `y` no se evalúa (necesariamente). Un patrón de explosión también puede aparecer en un constructor, por ejemplo,

```
data X = Con A !B C .. N
```

en cuyo caso se dice que el constructor `Con` es estricto en el campo `B` , lo que significa que el campo `B` se evalúa a WHNF cuando el constructor se aplica a argumentos suficientes (aquí, dos).

Patrones perezosos

Los patrones perezosos o *irrefutables* (indicados con syntax `~pat`) son patrones que siempre coinciden, sin siquiera mirar el valor coincidente. Esto significa que los patrones perezosos coincidirán incluso con los valores inferiores. Sin embargo, los usos subsiguientes de las variables vinculadas en sub-patrones de un patrón irrefutable forzarán la coincidencia del patrón a ocurrir, evaluándose al final a menos que la coincidencia sea exitosa.

La siguiente función es perezosa en su argumento:

```
f1 :: Either e Int -> Int
f1 ~(Right 1) = 42
```

y así conseguimos

```
λ» f1 (Right 1)
42
λ» f1 (Right 2)
42
λ» f1 (Left "foo")
42
λ» f1 (error "oops!")
42
λ» f1 "oops!"
*** type mismatch ***
```

La siguiente función está escrita con un patrón perezoso, pero de hecho está utilizando la variable del patrón que fuerza la coincidencia, por lo que fallará para los argumentos de la `Left` :

```
f2 :: Either e Int -> Int
f2 ~(Right x) = x + 1

λ» f2 (Right 1)
2
λ» f2 (Right 2)
3
λ» f2 (Right (error "oops!"))
*** Exception: oops!
λ» f2 (Left "foo")
*** Exception: lazypat.hs:5:1-21: Irrefutable pattern failed for pattern (Right x)
λ» f2 (error "oops!")
*** Exception: oops!
```

`let` enlaces sean perezosos, se comportan como patrones irrefutables:

```
act1 :: IO ()
act1 = do
    ss <- readLn
    let [s1, s2] = ss :: [String]
    putStrLn "Done"

act2 :: IO ()
act2 = do
    ss <- readLn
    let [s1, s2] = ss
    putStrLn s1
```

Aquí `act1` trabaja en entradas que se analizan en cualquier lista de cadenas, mientras que en `act2` el `putStrLn s1` necesita el valor de `s1` que fuerza la coincidencia del patrón para `[s1, s2]` , así que solo funciona para listas de exactamente dos cadenas:

```
λ» act1
> ["foo"]
Done
λ» act2
> ["foo"]
*** readIO: no parse ***
```

Campos estrictos

En una declaración de `data` , el prefijo de un tipo con un bang (`!`) Hace que el campo sea un *campo estricto* . Cuando se aplique el constructor de datos, esos campos se evaluarán con una forma normal de encabezado débil, por lo que se garantiza que los datos en los campos siempre estarán en forma normal de encabezado débil.

Los campos estrictos se pueden usar en los tipos de registro y no de registro:

```
data User = User
    { identifier :: !Int
    , firstName :: !Text
    , lastName :: !Text
    }

data T = MkT !Int !Int
```

Lea Rigor en línea: https://riptutorial.com/es/haskell/topic/3798/rigor

Capítulo 63: Sintaxis de grabación

Examples

Sintaxis basica

Los registros son una extensión del tipo de `data` algebraicos de suma que permiten nombrar campos:

```
data StandardType = StandardType String Int Bool --standard way to create a sum type

data RecordType = RecordType { -- the same sum type with record syntax
    aString :: String
  , aNumber :: Int
  , isTrue  :: Bool
  }
```

Los nombres de los campos se pueden usar para obtener el campo nombrado fuera del registro

```
> let r = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
> :t r
  r :: RecordType
> :t aString
  aString :: RecordType -> String
> aString r
  "Foobar"
```

Los registros pueden coincidir con el patrón

```
case r of
  RecordType{aNumber = x, aString=str} -> ... -- x = 42, str = "Foobar"
```

Observe que no es necesario nombrar todos los campos

Los registros se crean al nombrar sus campos, pero también se pueden crear como tipos de suma ordinarios (a menudo útiles cuando el número de campos es pequeño y no es probable que cambie)

```
r  = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
r' = RecordType  "Foobar" 42 True
```

Si se crea un registro sin un campo con nombre, el compilador emitirá una advertencia y el valor resultante será `undefined` .

```
> let r = RecordType {aString = "Foobar", aNumber= 42}
  <interactive>:1:9: Warning:
     Fields of RecordType not initialized: isTrue
> isTrue r
  Error 'undefined'
```

Un campo de un registro puede actualizarse estableciendo su valor. Los campos no mencionados no cambian.

```
> let r = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
> let r' = r{aNumber=117}
    -- r'{aString = "Foobar", aNumber= 117, isTrue = True}
```

A menudo es útil crear lentes para tipos de registros complicados.

Copiar registros al cambiar los valores de campo

Supongamos que tienes este tipo:

```
data Person = Person { name :: String, age:: Int } deriving (Show, Eq)
```

y dos valores:

```
alex = Person { name = "Alex", age = 21 }
jenny = Person { name = "Jenny", age = 36 }
```

se puede crear un nuevo valor de tipo `Person` copiando desde `alex` , especificando qué valores cambiar:

```
anotherAlex = alex { age = 31 }
```

Los valores de `alex` y `anotherAlex` ahora serán:

```
Person {name = "Alex", age = 21}

Person {name = "Alex", age = 31}
```

Graba con newtype

La sintaxis de registro se puede usar con `newtype` con la restricción de que hay exactamente un constructor con exactamente un campo. El beneficio aquí es la creación automática de una función para desenvolver el newtype. Estos campos a menudo se denominan comenzando con `run` para mónadas, `get` para monoides y `un` para otros tipos.

```
newtype State s a = State { runState :: s -> (s, a) }

newtype Product a = Product { getProduct :: a }

newtype Fancy = Fancy { unfancy :: String }
  -- a fancy string that wants to avoid concatenation with ordinary strings
```

Es importante tener en cuenta que la sintaxis de registro generalmente nunca se usa para formar valores y que el nombre del campo se usa estrictamente para desenvolver

```
getProduct $ mconcat [Product 7, Product 9, Product 12]
```

```
-- > 756
```

RecordWildCards

```
{-# LANGUAGE RecordWildCards #-}

data Client = Client { firstName     :: String
                     , lastName      :: String
                     , clientID      :: String
                     } deriving (Show)

printClientName :: Client -> IO ()
printClientName Client{..} = do
    putStrLn firstName
    putStrLn lastName
    putStrLn clientID
```

El patrón `Client{..}` trae en alcance todos los campos del constructor `Client` , y es equivalente al patrón

```
Client{ firstName = firstName, lastName = lastName, clientID = clientID }
```

También se puede combinar con otros igualadores de campo como son:

```
Client { firstName = "Joe", .. }
```

Esto es equivalente a

```
Client{ firstName = "Joe", lastName = lastName, clientID = clientID }
```

Definición de un tipo de datos con etiquetas de campo.

Es posible definir un tipo de datos con etiquetas de campo.

```
data Person = Person { age :: Int, name :: String }
```

Esta definición difiere de una definición de registro normal, ya que también define * los accesores de *registro* que se pueden usar para acceder a partes de un tipo de datos.

En este ejemplo, se definen dos accesores de registro, `age` y `name` , que nos permiten acceder a los campos de `age` y `name` respectivamente.

```
age :: Person -> Int
name :: Person -> String
```

Los accesores de registro son solo funciones de Haskell que son generadas automáticamente por el compilador. Como tales, se utilizan como funciones de Haskell ordinarias.

Al nombrar los campos, también podemos usar las etiquetas de campo en otros contextos para

hacer que nuestro código sea más legible.

La coincidencia de patrones

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name = x }) = map toLower x
```

Podemos vincular el valor ubicado en la posición de la etiqueta de campo relevante mientras el patrón se ajusta a un nuevo valor (en este caso `x`) que puede usarse en la RHS de una definición.

Coincidencia de patrones con `NamedFieldPuns`

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name }) = map toLower name
```

La extensión `NamedFieldPuns` lugar nos permite simplemente especificar la etiqueta de campo con la que queremos hacer coincidir, luego este nombre aparece sombreado en la RHS de una definición, por lo que referirse a `name` refiere al valor en lugar del acceso al registro.

Patrón a juego con `RecordWildcards`

```
lowerCaseName :: Person -> String
lowerCaseName (Person { .. }) = map toLower name
```

Cuando se hacen coincidencias con `RecordWildCards` , todas las etiquetas de campo se ponen en alcance. (En este ejemplo específico, `name` y `age`)

Esta extensión es un poco controvertida, ya que no está claro cómo se ponen los valores en el alcance si no está seguro de la definición de `Person` .

Actualizaciones de registro

```
setName :: String -> Person -> Person
setName newName person = person { name = newName }
```

También hay una sintaxis especial para actualizar los tipos de datos con etiquetas de campo.

Lea Sintaxis de grabación en línea: https://riptutorial.com/es/haskell/topic/1950/sintaxis-de-grabacion

Capítulo 64: Sintaxis en funciones

Examples

Guardias

Se puede definir una función utilizando guardas, que se puede pensar en clasificar el comportamiento de acuerdo con la entrada.

Tome la siguiente definición de función:

```
absolute :: Int -> Int  -- definition restricted to Ints for simplicity
absolute n = if (n < 0) then (-n) else n
```

Podemos reorganizarlo utilizando guardas:

```
absolute :: Int -> Int
absolute n
  | n < 0 = -n
  | otherwise = n
```

En este contexto, de lo `otherwise` es un alias significativo para `True` , por lo que siempre debe ser la última guardia.

La coincidencia de patrones

Haskell admite expresiones de coincidencia de patrones tanto en la definición de funciones como a través de declaraciones de `case` .

Una declaración de caso es muy similar a un interruptor en otros idiomas, excepto que admite todos los tipos de Haskell.

Comencemos simple:

```
longName :: String -> String
longName name = case name of
                  "Alex"  -> "Alexander"
                  "Jenny" -> "Jennifer"
                  _       -> "Unknown"  -- the "default" case, if you like
```

O bien, podríamos definir nuestra función como una ecuación que sería la coincidencia de patrones, sin usar una declaración de `case` :

```
longName "Alex"  = "Alexander"
longName "Jenny" = "Jennifer"
longName _       = "Unknown"
```

Un ejemplo más común es con el tipo `Maybe` :

```
data Person = Person { name :: String, petName :: (Maybe String) }

hasPet :: Person -> Bool
hasPet (Person _ Nothing) = False
hasPet _ = True  -- Maybe can only take `Just a` or `Nothing`, so this wildcard suffices
```

La coincidencia de patrones también se puede utilizar en listas:

```
isEmptyList :: [a] -> Bool
isEmptyList [] = True
isEmptyList _  = False

addFirstTwoItems :: [Int] -> [Int]
addFirstTwoItems []        = []
addFirstTwoItems (x:[])    = [x]
addFirstTwoItems (x:y:ys)  = (x + y) : ys
```

En realidad, la coincidencia de patrones se puede utilizar en cualquier constructor para cualquier clase de tipo. Por ejemplo, el constructor para listas es `:` y para tuplas `,`

Usando donde y guardias

Teniendo en cuenta esta función:

```
annualSalaryCalc :: (RealFloat a) => a -> a -> String
annualSalaryCalc hourlyRate weekHoursOfWork
  | hourlyRate * (weekHoursOfWork * 52) <= 40000  = "Poor child, try to get another job"
  | hourlyRate * (weekHoursOfWork * 52) <= 120000 = "Money, Money, Money!"
  | hourlyRate * (weekHoursOfWork * 52) <= 200000 = "Ri¢hie Ri¢h"
  | otherwise = "Hello Elon Musk!"
```

Podemos usar `where` evitar la repetición y hacer que nuestro código sea más legible. Vea la función alternativa a continuación, usando `where` :

```
annualSalaryCalc' :: (RealFloat a) => a -> a -> String
annualSalaryCalc' hourlyRate weekHoursOfWork
  | annualSalary <= smallSalary  = "Poor child, try to get another job"
  | annualSalary <= mediumSalary = "Money, Money, Money!"
  | annualSalary <= highSalary   = "Ri¢hie Ri¢h"
  | otherwise = "Hello Elon Musk!"
  where
      annualSalary = hourlyRate * (weekHoursOfWork * 52)
      (smallSalary, mediumSalary, highSalary)  = (40000, 120000, 200000)
```

Como se observó, utilizamos el punto `where` al final del cuerpo de la función, eliminamos la repetición del cálculo (`hourlyRate * (weekHoursOfWork * 52)`) y también usamos `where` organizar el rango de salario.

El nombramiento de sub-expresiones comunes también se puede lograr con la `let` expresiones, pero sólo el `where` la sintaxis hace posible que los *guardias* que se refieren a las sub-expresiones nombradas.

Lea Sintaxis en funciones en línea: https://riptutorial.com/es/haskell/topic/3799/sintaxis-en-

funciones

Capítulo 65: Solicitud parcial

Observaciones

Vamos a aclarar algunos conceptos erróneos que los principiantes pueden hacer.

Es posible que haya encontrado funciones tales como:

```
max :: (Ord a) => a -> a -> a
max m n
  | m >= n = m
  | otherwise = n
```

Los principiantes típicamente verán `max :: (Ord a) => a -> a -> a` como `max :: (Ord a) => a -> a -> a` función que toma dos argumentos (valores) de tipo `a` y devuelve un valor de tipo `a` . Sin embargo, lo que realmente está sucediendo es que `max` está **tomando un argumento** de tipo `a` y **devolviendo una función** de tipo `a -> a` . Esta función toma un argumento de tipo `a` y devuelve un valor final de tipo `a` .

De hecho, `max` puede escribirse como `max :: (Ord a) => a -> (a -> a)`

Considere la firma del tipo de `max` :

```
Prelude> :t max
max :: Ord a => a -> a -> a

Prelude> :t (max 75)
(max 75) :: (Num a, Ord a) => a -> a

Prelude> :t (max "Fury Road")
(max "Fury Road") :: [Char] -> [Char]

Prelude> :t (max "Fury Road" "Furiosa")
(max "Fury Road" "Furiosa") :: [Char]
```

`max 75` y `max "Fury Road"` puede que no *parezcan* funciones, pero en realidad lo son.

La confusión se deriva del hecho de que en matemáticas y en muchos otros lenguajes de programación comunes, se nos permite tener funciones que toman múltiples argumentos. Sin embargo, en Haskell, las **funciones solo pueden tomar un argumento** y pueden devolver valores como `a` , o funciones como `a -> a` .

Examples

Función de adición parcialmente aplicada

Podemos usar *la aplicación parcial* para "bloquear" el primer argumento. Después de aplicar un argumento, nos quedamos con una función que espera un argumento más antes de devolver el

resultado.

```
(+) :: Int -> Int -> Int

addOne :: Int -> Int
addOne = (+) 1
```

Luego podemos usar `addOne` para agregar uno a un `Int` .

```
> addOne 5
6
> map addOne [1,2,3]
[2,3,4]
```

Devolviendo una Función Parcialmente Aplicada

Devolver funciones parcialmente aplicadas es una técnica para escribir código conciso.

```
add :: Int -> Int -> Int
add x = (+x)

add 5 2
```

En este ejemplo (+ x) es una función parcialmente aplicada. Tenga en cuenta que el segundo parámetro de la función de adición no necesita ser especificado en la definición de la función.

El resultado de llamar a `add 5 2` es siete.

Secciones

La sección es una forma concisa de aplicar parcialmente argumentos a operadores de infijo.

Por ejemplo, si queremos escribir una función que agregue "ing" al final de una palabra, podemos usar una sección para definir sucintamente una función.

```
> (++ "ing") "laugh"
"laughing"
```

Observe cómo hemos aplicado parcialmente el segundo argumento. Normalmente, solo podemos aplicar parcialmente los argumentos en el orden especificado.

También podemos usar la sección izquierda para aplicar parcialmente el primer argumento.

```
> ("re" ++) "do"
"redo"
```

Podríamos escribir esto de manera equivalente usando la aplicación parcial de prefijo normal:

```
> ((++) "re") "do"
"redo"
```

Una nota sobre la resta

Los principiantes a menudo incorrectamente seccionan negación

```
> map (-1) [1,2,3]
***error: Could not deduce...
```

Esto no funciona, ya que `-1` se analiza como el literal `-1` lugar del operador seccionado `-` aplicado a `1` . La función `subtract` existe para evitar este problema.

```
> map (subtract 1) [1,2,3]
[0,1,2]
```

Lea Solicitud parcial en línea: https://riptutorial.com/es/haskell/topic/1954/solicitud-parcial

Capítulo 66: Streaming IO

Examples

Streaming IO

`io-streams` es una biblioteca basada en Stream que se centra en la abstracción de Stream, pero para IO. Expone dos tipos:

- `InputStream` : un controlador inteligente de solo lectura
- `OutputStream` : un controlador inteligente de solo escritura

Podemos crear un flujo con `makeInputStream :: IO (Maybe a) -> IO (InputStream a)` . La lectura de un flujo se realiza usando `read :: InputStream a -> IO (Maybe a)` , donde `Nothing` denota un EOF:

```
import Control.Monad (forever)
import qualified System.IO.Streams as S
import System.Random (randomRIO)

main :: IO ()
main = do
  is <- S.makeInputStream $ randomInt  -- create an InputStream
  forever $ printStream =<< S.read is  -- forever read from that stream
  return ()

randomInt :: IO (Maybe Int)
randomInt = do
  r <- randomRIO (1, 100)
  return $ Just r

printStream :: Maybe Int -> IO ()
printStream Nothing  = print "Nada!"
printStream (Just a) = putStrLn $ show a
```

Lea Streaming IO en línea: https://riptutorial.com/es/haskell/topic/4984/streaming-io

Capítulo 67: Tipo algebra

Examples

Números naturales en tipo álgebra

Podemos dibujar una conexión entre los tipos de Haskell y los números naturales. Esta conexión se puede hacer asignando a cada tipo el número de habitantes que tiene.

Tipos de union finita

Para tipos finitos, basta con ver que podemos asignar un tipo natural a cada número, basado en el número de constructores. Por ejemplo:

```
type Color = Red | Yellow | Green
```

sería **3** . Y el tipo `Bool` sería **2** .

```
type Bool = True | False
```

Unicidad hasta el isomorfismo

Hemos visto que múltiples tipos corresponderían a un solo número, pero en este caso, serían isomorfos. Es decir, habría un par de morfismos `f` y `g` , cuya composición sería la identidad, conectando los dos tipos.

```
f :: a -> b
g :: b -> a

f . g == id == g . f
```

En este caso, diríamos que los tipos son **isomorfos** . Consideraremos dos tipos iguales en nuestro álgebra siempre que sean isomorfos.

Por ejemplo, dos representaciones diferentes del número dos son trivalmente isomorfas:

```
type Bit  = I    | O
type Bool = True | False

bitValue :: Bit -> Bool
bitValue I = True
bitValue O = False

booleanBit :: Bool -> Bit
booleanBit True  = I
booleanBit False = O
```

Porque podemos ver `bitValue . booleanBit == id == booleanBit . bitValue`

Uno y cero

La representación del número **1** es obviamente un tipo con un solo constructor. En Haskell, este tipo es canónicamente el tipo `()` , llamado Unidad. Cada otro tipo con un solo constructor es isomorfo a `()` .

Y nuestra representación de **0** será un tipo sin constructores. Este es el tipo **Void** en Haskell, como se define en `Data.Void` . Esto sería equivalente a un tipo deshabitado, sin constructores de datos:

```
data Void
```

Suma y multiplicación

La suma y multiplicación tienen equivalentes en este tipo de álgebra. Se corresponden con las **uniones etiquetadas** y **tipos de producto** .

```
data Sum a b = A a | B b
data Prod a b = Prod a b
```

Podemos ver cómo el número de habitantes de cada tipo corresponde a las operaciones del álgebra.

De manera equivalente, podemos usar `Either` y `(,)` como constructores de tipo para la suma y la multiplicación. Son isomorfos a nuestros tipos previamente definidos:

```
type Sum' a b = Either a b
type Prod' a b = (a,b)
```

Los resultados esperados de la suma y la multiplicación son seguidos por el tipo de álgebra hasta el isomorfismo. Por ejemplo, podemos ver un isomorfismo entre 1 + 2, 2 + 1 y 3; como 1 + 2 = 3 = 2 + 1.

```
data Color = Red | Green | Blue

f :: Sum () Bool -> Color
f (Left ())     = Red
f (Right True)  = Green
f (Right False) = Blue

g :: Color -> Sum () Bool
g Red   = Left ()
g Green = Right True
g Blue  = Right False

f' :: Sum Bool () -> Color
f' (Right ())   = Red
```

```
f' (Left True)  = Green
f' (Left False) = Blue

g' :: Color -> Sum Bool ()
g' Red   = Right ()
g' Green = Left True
g' Blue  = Left False
```

Reglas de suma y multiplicación.

Las reglas comunes de conmutatividad, asociatividad y distributividad son válidas porque existen isomorfismos triviales entre los siguientes tipos:

```
-- Commutativity
Sum a b           <=> Sum b a
Prod a b          <=> Prod b a
-- Associativity
Sum (Sum a b) c   <=> Sum a (Sum b c)
Prod (Prod a b) c <=> Prod a (Prod b c)
-- Distributivity
Prod a (Sum b c)  <=> Sum (Prod a b) (Prod a c)
```

Tipos recursivos

Liza

Las listas se pueden definir como:

```
data List a = Nil | Cons a (List a)
```

Si traducimos esto a nuestro tipo de álgebra, obtenemos

> Lista (a) = 1 + a * Lista (a)

Pero ahora podemos sustituir la *Lista (a) de* nuevo en esta expresión varias veces, para obtener:

> Lista (a) = 1 + a + a * a + a * a * a + a * a * a * a + ...

Esto tiene sentido si vemos una lista como un tipo que puede contener solo un valor, como en `[]` ; o cada valor de tipo `a` , como en `[x]` ; o dos valores de tipo `a` , como en `[x,y]` ; y así. La definición teórica de Lista que deberíamos obtener de allí sería:

```
-- Not working Haskell code!
data List a = Nil
            | One a
            | Two a a
            | Three a a a
            ...
```

Arboles

Podemos hacer lo mismo con árboles binarios, por ejemplo. Si los definimos como:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Obtenemos la expresión:

> Árbol (a) = 1 + a * Árbol (a) * Árbol (a)

Y si hacemos las mismas sustituciones una y otra vez, obtendríamos la siguiente secuencia:

> Árbol (a) = 1 + a + 2 (a * a) + 5 (a * a * a) + 14 (a * a * a * a) + ...

Los coeficientes que obtenemos aquí corresponden a la secuencia de números catalanes, y el número catalan n-th es precisamente el número de árboles binarios posibles con n nodos.

Derivados

La derivada de un tipo es el tipo de su tipo de contextos de un agujero. Este es el tipo que obtendríamos si hacemos que una variable de tipo desaparezca en cada punto posible y sumemos los resultados.

Como ejemplo, podemos tomar el tipo triple `(a,a,a)` , y derivarlo, obteniendo

```
data OneHoleContextsOfTriple = (a,a,()) | (a,(),a) | ((),a,a)
```

Esto es coherente con nuestra definición habitual de derivación, como:

> d / da (a * a * a) = 3 * a * a

Más sobre este tema se puede leer en este artículo .

Funciones

Las funciones pueden verse como exponenciales en nuestro álgebra. Como podemos ver, si tomamos un tipo `a` con n instancias y un tipo `b` con m instancias, el tipo `a -> b` tendrá m para el poder de n instancias.

Como ejemplo, `Bool -> Bool` es isomorfo a `(Bool,Bool)` , como 2 * 2 = 2^2.

```
iso1 :: (Bool -> Bool) -> (Bool,Bool)
iso1 f = (f True,f False)

iso2 :: (Bool,Bool) -> (Bool -> Bool)
iso2 (x,y) = (\p -> if p then x else y)
```

Lea Tipo algebra en línea: https://riptutorial.com/es/haskell/topic/4905/tipo-algebra

Capítulo 68: Tipo de solicitud

Introducción

`TypeApplications` son una alternativa a las *anotaciones de* tipo cuando el compilador `TypeApplications` inferir tipos para una expresión dada.

Esta serie de ejemplos explicará el propósito de la extensión `TypeApplications` y cómo usarla

No olvide habilitar la extensión colocando `{-# LANGUAGE TypeApplications #-}` en la parte superior de su archivo fuente.

Examples

Evitar anotaciones de tipo.

Utilizamos anotaciones de tipo para evitar la ambigüedad. Las aplicaciones de tipo se pueden utilizar para el mismo propósito. Por ejemplo

```
x :: Num a => a
x = 5

main :: IO ()
main = print x
```

Este código tiene un error de ambigüedad. Sabemos que `a` tiene una instancia de `Num` , y para imprimirla sabemos que necesita una instancia de `Show` . Esto podría funcionar si `a` era, por ejemplo, un `Int` , así que para corregir el error podemos agregar una anotación de tipo

```
main = print (x :: Int)
```

Otra solución usando aplicaciones de tipo se vería así

```
main = print @Int x
```

Para entender lo que esto significa, debemos observar el tipo de firma de `print` .

```
print :: Show a => a -> IO ()
```

La función toma un parámetro de tipo `a` , pero otra forma de verlo es que en realidad toma dos parámetros. El primero es un parámetro de *tipo* , el segundo es un valor cuyo tipo es el primer parámetro.

La principal diferencia entre los parámetros de valor y los parámetros de tipo es que los últimos se proporcionan de forma implícita a las funciones cuando los llamamos. ¿Quién los provee? El algoritmo de inferencia de tipos! Lo que `TypeApplications` nos permite hacer es dar a esos tipos de

parámetros explícitamente. Esto es especialmente útil cuando la inferencia de tipo no puede determinar el tipo correcto.

Así que para romper el ejemplo anterior

```
print :: Show a => a -> IO ()
print @Int :: Int -> IO ()
print @Int x :: IO ()
```

Escriba aplicaciones en otros idiomas

Si está familiarizado con lenguajes como Java, C # o C ++ y el concepto de genéricos / plantillas, esta comparación puede ser útil para usted.

Digamos que tenemos una función genérica en C #

```
public static T DoNothing<T>(T in) { return in; }
```

Para llamar a esta función con un `float` podemos hacer `DoNothing(5.0f)` o si queremos ser explícitos podemos decir `DoNothing<float>(5.0f)` . Esa parte dentro de los soportes de ángulo es la aplicación de tipo.

En Haskell es lo mismo, excepto que los parámetros de tipo no solo son implícitos en los sitios de llamadas sino también en los sitios de definición.

```
doNothing :: a -> a
doNothing x = x
```

Esto también se puede hacer explícito usando las `ScopedTypeVariables` , `Rank2Types` o `RankNTypes` como esta.

```
doNothing :: forall a. a -> a
doNothing x = x
```

Luego, en el sitio de la llamada, podemos volver a escribir `doNothing 5.0` o `doNothing @Float 5.0`

Orden de parametros

El problema de que los argumentos de tipo sean implícitos se vuelve obvio una vez que tenemos más de uno. ¿En qué orden entran?

```
const :: a -> b -> a
```

¿Escribir `const @Int` significa que `a` es igual a `Int` , o es `b` ? En caso de que indiquemos explícitamente los parámetros de tipo usando `forall` `const :: forall a b. a -> b -> a` `forall` como `const :: forall a b. a -> b -> a` entonces el orden es como está escrito: `a` , luego `b` .

Si no lo hacemos, entonces el orden de las variables es de izquierda a derecha. La primera

variable que se debe mencionar es el primer parámetro de tipo, la segunda es el segundo parámetro de tipo y así sucesivamente.

¿Qué pasa si queremos especificar la segunda variable de tipo, pero no la primera? Podemos usar un comodín para la primera variable como esta

```
const @_ @Int
```

El tipo de esta expresión es

```
const @_ @Int :: a -> Int -> a
```

Interacción con tipos ambiguos.

Digamos que estás introduciendo una clase de tipos que tienen un tamaño en bytes.

```
class SizeOf a where
    sizeOf :: a -> Int
```

El problema es que el tamaño debe ser constante para cada valor de ese tipo. En realidad, no queremos que la función `sizeOf` dependa de `a` , sino solo de su tipo.

Sin aplicaciones de tipo, la mejor solución que tuvimos fue el tipo de `Proxy` definido de esta manera

```
data Proxy a = Proxy
```

El propósito de este tipo es llevar información de tipo, pero no información de valor. Entonces nuestra clase podría verse así.

```
class SizeOf a where
    sizeOf :: Proxy a -> Int
```

Ahora te estarás preguntando, ¿por qué no descartar el primer argumento por completo? El tipo de nuestra función sería entonces `sizeOf :: Int` o, para ser más precisos porque es un método de una clase, `sizeOf :: SizeOf a => Int` o para ser aún más explícito `sizeOf :: forall a. SizeOf a => Int` .

El problema es la inferencia de tipos. Si escribo `sizeOf` algún lugar, el algoritmo de inferencia solo sabe que espero un `Int` . No tiene idea de qué tipo quiero sustituir por `a` . Debido a esto, la definición es rechazada por el compilador a *menos* que tenga habilitada la extensión `{-# LANGUAGE AllowAmbiguousTypes #-}` . En ese caso, la definición compila, simplemente no puede usarse en ningún lugar sin un error de ambigüedad.

Afortunadamente, la introducción de aplicaciones de tipo salva el día! Ahora podemos escribir `sizeOf @Int` , diciendo explícitamente que `a` es `Int` . Las aplicaciones de tipo nos permiten proporcionar un parámetro de tipo, incluso si no aparece en los *parámetros reales de la función* .

Lea Tipo de solicitud en línea: https://riptutorial.com/es/haskell/topic/10767/tipo-de-solicitud

Capítulo 69: Tipo Familias

Examples

Tipo de familias de sinónimos

Las familias de sinónimos de tipo son solo funciones de nivel de tipo: asocian tipos de parámetros con tipos de resultados. Estos vienen en tres variedades diferentes.

Familias de sinónimos de tipo cerrado

Estos funcionan de manera muy similar a las funciones Haskell de nivel de valor ordinarias: usted especifica algunas cláusulas, asignando ciertos tipos a otros:

```
{-# LANGUAGE TypeFamilies #-}
type family Vanquisher a where
    Vanquisher Rock = Paper
    Vanquisher Paper = Scissors
    Vanquisher Scissors = Rock

data Rock=Rock; data Paper=Paper; data Scissors=Scissors
```

Abrir familias de sinónimos de tipo

Estos funcionan más como instancias de typeclass: cualquiera puede agregar más cláusulas en otros módulos.

```
type family DoubledSize w

type instance DoubledSize Word16 = Word32
type instance DoubledSize Word32 = Word64
-- Other instances might appear in other modules, but two instances cannot overlap
-- in a way that would produce different results.
```

Sinónimos asociados a clases

Una familia de tipo abierto también se puede combinar con una clase real. Esto generalmente se hace cuando, al igual que con las familias de datos asociadas , algún método de clase necesita objetos auxiliares adicionales, y estos objetos auxiliares *pueden* ser diferentes para diferentes instancias, pero posiblemente también se pueden compartir. Un buen ejemplo es la clase `VectorSpace` :

```
class VectorSpace v where
  type Scalar v :: *
```

```
  (*^) :: Scalar v -> v -> v

instance VectorSpace Double where
  type Scalar Double = Double
  μ *^ n = μ * n

instance VectorSpace (Double,Double) where
  type Scalar (Double,Double) = Double
  μ *^ (n,m) = (μ*n, μ*m)

instance VectorSpace (Complex Double) where
  type Scalar (Complex Double) = Complex Double
  μ *^ n = μ*n
```

Observe cómo en las dos primeras instancias, la implementación de `Scalar` es la misma. Esto no sería posible con una familia de datos asociada: las familias de datos son inyectivas , pero las familias sinónimo de tipo no lo son.

Si bien la no inyectividad abre algunas posibilidades como la anterior, también dificulta la inferencia de tipos. Por ejemplo, lo siguiente no verificará:

```
class Foo a where
  type Bar a :: *
  bar :: a -> Bar a
instance Foo Int where
  type Bar Int = String
  bar = show
instance Foo Double where
  type Bar Double = Bool
  bar = (>0)

main = putStrLn (bar 1)
```

En este caso, el compilador no puede saber qué instancia usar, porque el argumento de la `bar` es en sí mismo un literal `Num` polimórfico. Y la función de `Bar` tipo no se puede resolver en "dirección inversa", precisamente porque no es inyectiva † y, por lo tanto, no es invertible (podría haber más de un tipo con `Bar a = String`).

---

† Con solo estas dos instancias, en realidad *es* inyectivo, pero el compilador no puede saber que alguien no agregará más instancias más adelante y, por lo tanto, romperá el comportamiento.

Familias de tipos de datos

Las familias de datos se pueden usar para construir tipos de datos que tienen diferentes implementaciones basadas en sus argumentos de tipo.

Familias de datos independientes

```
{-# LANGUAGE TypeFamilies #-}
data family List a
data instance List Char = Nil | Cons Char (List Char)
```

```
data instance List () = UnitList Int
```

En la declaración anterior, `Nil :: List Char` y `UnitList :: Int -> List ()`

Familias de datos asociados

Las familias de datos también pueden asociarse con typeclasses. Esto suele ser útil para los tipos con "objetos auxiliares", que son necesarios para los métodos de clase de tipos genéricos, pero deben contener información diferente según la instancia concreta. Por ejemplo, las ubicaciones de indexación en una lista solo requieren un solo número, mientras que en un árbol necesita un número para indicar la ruta en cada nodo:

```
class Container f where
  data Location f
  get :: Location f -> f a -> Maybe a

instance Container [] where
  data Location [] = ListLoc Int
  get (ListLoc i) xs
    | i < length xs  = Just $ xs!!i
    | otherwise      = Nothing

instance Container Tree where
  data Location Tree = ThisNode | NodePath Int (Location Tree)
  get ThisNode (Node x _) = Just x
  get (NodePath i path) (Node _ sfo) = get path =<< get i sfo
```

La inyectividad

Tipo Las familias no son necesariamente inyectivas. Por lo tanto, no podemos inferir el parámetro de una aplicación. Por ejemplo, en `servant` , dado un `Server a` tipo `Server a` no podemos inferir el tipo `a` . Para resolver este problema, podemos utilizar `Proxy` . Por ejemplo, en el `servant` , la función de `serve` tiene tipo `... Proxy a -> Server a -> ...` Podemos inferir `a` de `Proxy a` porque `Proxy` está definido por `data` que son inyectivos.

Lea Tipo Familias en línea: https://riptutorial.com/es/haskell/topic/2955/tipo-familias

Capítulo 70: Tipos de datos algebraicos generalizados

Examples

Uso básico

Cuando la extensión `GADTs` está habilitada, además de las declaraciones de datos regulares, también puede declarar tipos de datos algebraicos generalizados de la siguiente manera:

```
data DataType a where
    Constr1 :: Int -> a -> Foo a -> DataType a
    Constr2 :: Show a => a -> DataType a
    Constr3 :: DataType Int
```

Una declaración GADT enumera los tipos de todos los constructores que tiene un tipo de datos, explícitamente. A diferencia de las declaraciones de tipos de datos regulares, el tipo de un constructor puede ser cualquier función N-ary (incluyendo nullary) que finalmente resulte en el tipo de datos aplicado a algunos argumentos.

En este caso, hemos declarado que el tipo `DataType` tiene tres constructores: `Constr1` , `Constr2` y `Constr3` .

El constructor `Constr1` no es diferente de uno declarado utilizando una declaración de datos regular: `data DataType a = Constr1 Int a (Foo a) | ...`

`Constr2` embargo, `Constr2` requiere que `a` tenga una instancia de `Show` , por lo tanto, al usar el constructor, la instancia debería existir. Por otro lado, cuando la concordancia de patrones en él, el hecho de que `a` es una instancia de `Show` entra en su alcance, para que pueda escribir:

```
foo :: DataType a -> String
foo val = case val of
    Constr2 x -> show x
    ...
```

Tenga en cuenta que la función `Show a` restricción no aparece en el tipo de la función, y solo está visible en el código a la derecha de `->` .

`Constr3` tiene el tipo `DataType Int` , lo que significa que cada vez que un valor del tipo `DataType a` es un `Constr3` , se sabe que `a ~ Int` . Esta información también se puede recuperar con una coincidencia de patrón.

Lea Tipos de datos algebraicos generalizados en línea:
https://riptutorial.com/es/haskell/topic/2971/tipos-de-datos-algebraicos-generalizados

Capítulo 71: Tipos fantasma

Examples

Caso de uso para tipos fantasmas: Monedas

Los tipos fantasma son útiles para tratar con datos, que tienen representaciones idénticas pero no son lógicamente del mismo tipo.

Un buen ejemplo es tratar con monedas. Si trabaja con monedas, absolutamente nunca querrá, por ejemplo, agregar dos cantidades de monedas diferentes. ¿Cuál sería la divisa resultante de `5.32€ + 2.94$` ? No está definido y no hay una buena razón para hacer esto.

Una solución para esto podría ser algo como esto:

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

data USD
data EUR

newtype Amount a = Amount Double
                 deriving (Show, Eq, Ord, Num)
```

La extensión `GeneralisedNewtypeDeriving` nos permite derivar `Num` para el tipo de `Amount` . GHC reutiliza la instancia de `Double` 's `Num` .

Ahora, si representa cantidades en euros con, por ejemplo, `(5.0 :: Amount EUR)` , ha resuelto el problema de mantener las cantidades dobles separadas en el nivel de tipo sin introducir gastos generales. Cosas como `(1.13 :: Amount EUR) + (5.30 :: Amount USD)` resultarán en un error de tipo y requerirán que trate con la conversión de moneda de manera apropiada.

Se puede encontrar documentación más completa en el artículo de haskell wiki.

Lea Tipos fantasma en línea: https://riptutorial.com/es/haskell/topic/5227/tipos-fantasma

Capítulo 72: Transformadores de mónada

Examples

Un contador monádico

Un ejemplo sobre cómo componer el lector, el escritor y la mónada de estado utilizando transformadores de mónada. El código fuente se puede encontrar en este repositorio.

Queremos implementar un contador, que incremente su valor en una constante dada.

Comenzamos definiendo algunos tipos y funciones:

```
newtype Counter = MkCounter {cValue :: Int}
  deriving (Show)

-- | 'inc c n' increments the counter by 'n' units.
inc :: Counter -> Int -> Counter
inc (MkCounter c) n = MkCounter (c + n)
```

Supongamos que queremos realizar el siguiente cálculo utilizando el contador:

- poner el contador a 0
- establece la constante de incremento a 3
- incrementar el contador 3 veces
- establece la constante de incremento a 5
- incrementar el contador 2 veces

La mónada estatal proporciona abstracciones para pasar el estado. Podemos hacer uso de la mónada de estado y definir nuestra función de incremento como un transformador de estado.

```
-- | CounterS is a monad.
type CounterS = State Counter

-- | Increment the counter by 'n' units.
incS :: Int-> CounterS ()
incS n = modify (\c -> inc c n)
```

Esto ya nos permite expresar un cálculo de una manera más clara y sucinta:

```
-- | The computation we want to run, with the state monad.
mComputationS :: CounterS ()
mComputationS = do
  incS 3
  incS 3
  incS 3
  incS 5
  incS 5
```

Pero todavía tenemos que pasar la constante de incremento en cada invocación. Nos gustaría

evitar esto.

Añadiendo un entorno

La mónada del lector proporciona una forma conveniente de pasar un entorno. Esta mónada se usa en la programación funcional para realizar lo que en el mundo OO se conoce como *inyección de dependencia* .

En su versión más simple, la mónada del lector requiere dos tipos:

- el tipo de valor que se lee (es decir, nuestro entorno, `r` continuación),
- el valor devuelto por la mónada lector (`a` continuación).

  Lector ra

Sin embargo, también necesitamos hacer uso de la mónada estatal. Por lo tanto, necesitamos usar el transformador `ReaderT` :

```
newtype ReaderT r m a :: * -> (* -> *) -> * -> *
```

Usando `ReaderT` , podemos definir nuestro contador con el entorno y el estado de la siguiente manera:

```
type CounterRS = ReaderT Int CounterS
```

Definimos una función `incR` que toma la constante de incremento del entorno (utilizando `ask`), y para definir nuestra función de incremento en términos de nuestra mónada `CounterS` utilizamos la función de `lift` (que pertenece a la clase de transformador de mónada).

```
-- | Increment the counter by the amount of units specified by the environment.
incR :: CounterRS ()
incR = ask >>= lift . incS
```

Usando la mónada del lector podemos definir nuestro cálculo de la siguiente manera:

```
-- | The computation we want to run, using reader and state monads.
mComputationRS :: CounterRS ()
mComputationRS = do
  local (const 3) $ do
    incR
    incR
    incR
    local (const 5) $ do
      incR
      incR
```

Los requisitos cambiaron: necesitamos

registro!

Ahora suponga que queremos agregar el registro a nuestro cálculo, de modo que podamos ver la evolución de nuestro contador a tiempo.

También tenemos una mónada para realizar esta tarea, la mónada escritora . Al igual que con la mónada del lector, ya que los estamos componiendo, necesitamos hacer uso del transformador de mónada del lector:

```
newtype WriterT w m a :: * -> (* -> *) -> * -> *
```

Aquí `w` representa el tipo de salida a acumular (que debe ser un monoide, que nos permite acumular este valor), `m` es la mónada interna y `a` tipo de cálculo.

Luego podemos definir nuestro contador con registro, entorno y estado de la siguiente manera:

```
type CounterWRS = WriterT [Int] CounterRS
```

Y haciendo uso de `lift` podemos definir la versión de la función de incremento que registra el valor del contador después de cada incremento:

```
incW :: CounterWRS ()
incW = lift incR >> get >>= tell . (:[]) . cValue
```

Ahora el cálculo que contiene el registro se puede escribir de la siguiente manera:

```
mComputationWRS :: CounterWRS ()
mComputationWRS = do
  local (const 3) $ do
    incW
    incW
    incW
    local (const 5) $ do
      incW
      incW
```

Haciendo todo de una vez

Este ejemplo pretende mostrar los transformadores de mónada en funcionamiento. Sin embargo, podemos lograr el mismo efecto al componer todos los aspectos (entorno, estado y registro) en una sola operación de incremento.

Para ello hacemos uso de restricciones de tipo:

```
inc' :: (MonadReader Int m, MonadState Counter m, MonadWriter [Int] m) => m ()
inc' = ask >>= modify . (flip inc) >> get >>= tell . (:[]) . cValue
```

Aquí llegamos a una solución que funcionará para cualquier mónada que cumpla con las restricciones anteriores. La función de cálculo se define así con el tipo:

```
mComputation' :: (MonadReader Int m, MonadState Counter m, MonadWriter [Int] m) => m ()
```

ya que en su cuerpo hacemos uso de inc '.

Podríamos ejecutar este cálculo, en el REPL de `ghci` por ejemplo, de la siguiente manera:

```
runState ( runReaderT ( runWriterT mComputation' ) 15 )  (MkCounter 0)
```

Lea Transformadores de mónada en línea:
https://riptutorial.com/es/haskell/topic/7752/transformadores-de-monada

Capítulo 73: Tubería

Observaciones

Como la página de hackage describe:

> pipe es una biblioteca de procesamiento de secuencias limpia y potente que le permite crear y conectar componentes de transmisión reutilizables

Los programas implementados a través de la transmisión a menudo pueden ser sucintos y compositivos, con funciones cortas y simples que le permiten "encajar o sacar" fácilmente las funciones con el respaldo del sistema de tipo Haskell.

`await :: Monad m => Consumer' ama`

Extrae un valor del flujo ascendente, donde `a` es nuestro tipo de entrada.

`yield :: Monad m => a -> Producer' am ()`

Produce un valor, donde `a` es el tipo de salida.

Es muy recomendable que lea el paquete de `Pipes.Tutorial` incorporado que ofrece una excelente visión general de los conceptos básicos de Pipes y cómo interactúan `Producer` , `Consumer` y `Effect` .

Examples

Productores

Un `Producer` es una acción monádica que puede `yield` valores para el consumo posterior:

```
type Producer b = Proxy X () () b
yield :: Monad m => a -> Producer a m ()
```

Por ejemplo:

```
naturals :: Monad m => Producer Int m ()
naturals = each [1..] -- each is a utility function exported by Pipes
```

Por supuesto, podemos tener `Producer` que son funciones de otros valores también:

```
naturalsUntil :: Monad m => Int -> Producer Int m ()
naturalsUntil n = each [1..n]
```

Los consumidores

Un `Consumer` solo puede `await` valores de upstream.

```
type Consumer a = Proxy () a () X
await :: Monad m => Consumer a m a
```

Por ejemplo:

```
fancyPrint :: MonadIO m => Consumer String m ()
fancyPrint = forever $ do
  numStr <- await
  liftIO $ putStrLn ("I received: " ++ numStr)
```

Tubería

Las tuberías pueden `await` y `yield` .

```
type Pipe a b = Proxy () a () b
```

Este Pipe espera un `Int` y lo convierte en una `String` :

```
intToStr :: Monad m => Pipe Int String m ()
intToStr = forever $ await >>= (yield . show)
```

Corriendo tuberías con runEffect

Usamos `runEffect` para ejecutar nuestro `Pipe` :

```
main :: IO ()
main = do
  runEffect $ naturalsUntil 10 >-> intToStr >-> fancyPrint
```

Tenga en cuenta que `runEffect` requiere un `Effect` , que es un `Proxy` autónomo sin entradas ni salidas:

```
runEffect :: Monad m => Effect m r -> m r
type Effect = Proxy X () () X
```

(donde `X` es el tipo vacío, también conocido como `Void`).

Tubos de conexión

Use `>->` para conectar `Producer` s, `Consumer` s y `Pipe` s para componer funciones de `Pipe` más grandes.

```
printNaturals :: MonadIO m => Effect m ()
printNaturals = naturalsUntil 10 >-> intToStr >-> fancyPrint
```

`Consumer` tipos `Producer` , `Consumer` , `Pipe` y `Effect` se definen en términos del tipo de `Proxy` general. Por lo tanto, `>->` se puede utilizar para una variedad de propósitos. Los tipos definidos por el argumento de la izquierda deben coincidir con el tipo consumido por el argumento de la derecha:

```
(>->) :: Monad m => Producer b m r -> Consumer b   m r -> Effect       m r
(>->) :: Monad m => Producer b m r -> Pipe     b c m r -> Producer   c m r
(>->) :: Monad m => Pipe   a b m r -> Consumer b   m r -> Consumer a   m r
(>->) :: Monad m => Pipe   a b m r -> Pipe     b c m r -> Pipe     a c m r
```

El transformador proxy mónada

El tipo de datos central de las `pipes` es el transformador de mónada `Proxy` . `Pipe` , `Producer` , `Consumer` , etc. se definen en términos de `Proxy` .

Como `Proxy` es un transformador de mónadas, las definiciones de `Pipe` s toman la forma de scripts monádicos que `await` y `yield` valores, además de realizar efectos desde la mónada base `m` .

Combinación de tuberías y comunicación en red.

Pipes admite la comunicación binaria simple entre un cliente y un servidor

En este ejemplo:

1. un cliente se conecta y envía un `FirstMessage`
2. El servidor recibe y responde `DoSomething 0`
3. El cliente recibe y responde `DoNothing`
4. Los pasos 2 y 3 se repiten indefinidamente.

El tipo de datos de comando intercambiado a través de la red:

```
-- Command.hs
{-# LANGUAGE DeriveGeneric #-}
module Command where
import Data.Binary
import GHC.Generics (Generic)

data Command = FirstMessage
             | DoNothing
             | DoSomething Int
             deriving (Show,Generic)

instance Binary Command
```

Aquí, el servidor espera a que un cliente se conecte:

```
module Server where

import Pipes
import qualified Pipes.Binary as PipesBinary
import qualified Pipes.Network.TCP as PNT
import qualified Command as C
import qualified Pipes.Parse as PP
import qualified Pipes.Prelude as PipesPrelude

pageSize :: Int
pageSize = 4096

-- pure handler, to be used with PipesPrelude.map
```

```
pureHandler :: C.Command -> C.Command
pureHandler c = c  -- answers the same command that we have receveid

-- impure handler, to be used with PipesPremude.mapM
sideffectHandler :: MonadIO m => C.Command -> m C.Command
sideffectHandler c = do
  liftIO $ putStrLn $ "received message = " ++ (show c)
  return $ C.DoSomething 0
  -- whatever incoming command `c` from the client, answer DoSomething 0

main :: IO ()
main = PNT.serve (PNT.Host "127.0.0.1") "23456" $
  \(connectionSocket, remoteAddress) -> do
                putStrLn $ "Remote connection from ip = " ++ (show remoteAddress)
                _ <- runEffect $ do
                  let bytesReceiver = PNT.fromSocket connectionSocket pageSize
                  let commandDecoder = PP.parsed PipesBinary.decode bytesReceiver
                  commandDecoder >-> PipesPrelude.mapM sideffectHandler >-> for cat
PipesBinary.encode >-> PNT.toSocket connectionSocket
                  -- if we want to use the pureHandler
                  --commandDecoder >-> PipesPrelude.map pureHandler >-> for cat
PipesBinary.Encode >-> PNT.toSocket connectionSocket
                return ()
```

El cliente se conecta así:

```
module Client where

import Pipes
import qualified Pipes.Binary as PipesBinary
import qualified Pipes.Network.TCP as PNT
import qualified Pipes.Prelude as PipesPrelude
import qualified Pipes.Parse as PP
import qualified Command as C

pageSize :: Int
pageSize = 4096

-- pure handler, to be used with PipesPrelude.amp
pureHandler :: C.Command -> C.Command
pureHandler c = c  -- answer the same command received from the server

-- inpure handler, to be used with PipesPremude.mapM
sideffectHandler :: MonadIO m => C.Command -> m C.Command
sideffectHandler c = do
  liftIO $ putStrLn $ "Received: " ++ (show c)
  return C.DoNothing  -- whatever is received from server, answer DoNothing

main :: IO ()
main = PNT.connect ("127.0.0.1") "23456" $
  \(connectionSocket, remoteAddress) -> do
    putStrLn $ "Connected to distant server ip = " ++ (show remoteAddress)
    sendFirstMessage connectionSocket
    _ <- runEffect $ do
      let bytesReceiver = PNT.fromSocket connectionSocket pageSize
      let commandDecoder = PP.parsed PipesBinary.decode bytesReceiver
      commandDecoder >-> PipesPrelude.mapM sideffectHandler >-> for cat PipesBinary.encode >->
PNT.toSocket connectionSocket
    return ()
```

```
sendFirstMessage :: PNT.Socket -> IO ()
sendFirstMessage s = do
  _ <- runEffect $ do
    let encodedProducer = PipesBinary.encode C.FirstMessage
    encodedProducer >-> PNT.toSocket s
  return ()
```

Lea Tubería en línea: https://riptutorial.com/es/haskell/topic/6768/tuberia

Capítulo 74: Tuplas (pares, triples, ...)

Observaciones

- Haskell no admite tuplas con un componente de forma nativa.
- Las unidades (escritas `()`) pueden entenderse como tuplas con cero componentes.
- No hay funciones predefinidas para extraer componentes de tuplas con más de dos componentes. Si cree que necesita dichas funciones, considere usar un tipo de datos personalizado con etiquetas de registro en lugar del tipo de tupla. Luego puede usar las etiquetas de registro como funciones para extraer los componentes.

Examples

Construir valores de tupla

Usa paréntesis y comas para crear tuplas. Usa una coma para crear un par.

```
(1, 2)
```

Usa más comas para crear tuplas con más componentes.

```
(1, 2, 3)

(1, 2, 3, 4)
```

Tenga en cuenta que también es posible declarar las tuplas en su forma no saturada.

```
(,) 1 2     -- equivalent to (1,2)
(,,) 1 2 3  -- equivalent to (1,2,3)
```

Las tuplas pueden contener valores de diferentes tipos.

```
("answer", 42, '?')
```

Las tuplas pueden contener valores complejos como listas o más tuplas.

```
([1, 2, 3], "hello", ('A', 65))

(1, (2, (3, 4), 5), 6)
```

Escribe tipos de tuplas

Usa paréntesis y comas para escribir tipos de tuplas. Usa una coma para escribir un tipo de par.

```
(Int, Int)
```

Usa más comas para escribir tipos de tuplas con más componentes.

```
(Int, Int, Int)

(Int, Int, Int, Int)
```

Las tuplas pueden contener valores de diferentes tipos.

```
(String, Int, Char)
```

Las tuplas pueden contener valores complejos como listas o más tuplas.

```
([Int], String, (Char, Int))

(Int, (Int, (Int, Int), Int), Int)
```

Patrón de coincidencia en las tuplas

La coincidencia de patrones en las tuplas utiliza los constructores de tuplas. Para hacer coincidir un par, por ejemplo, usaríamos el constructor `(,)` :

```
myFunction1 (a, b) = ...
```

Usamos más comas para hacer coincidir las tuplas con más componentes:

```
myFunction2 (a, b, c) = ...

myFunction3 (a, b, c, d) = ...
```

Los patrones de tupla pueden contener patrones complejos, como los patrones de lista o más patrones de tupla.

```
myFunction4 ([a, b, c], d, e) = ...

myFunction5 (a, (b, (c, d), e), f) = ...
```

Extraer componentes de la tupla.

Utilice las funciones `fst` y `snd` (de `Prelude` o `Data.Tuple`) para extraer el primer y segundo componente de los pares.

```
fst (1, 2) -- evaluates to 1

snd (1, 2) -- evaluates to 2
```

O utilice la coincidencia de patrones.

```
case (1, 2) of (result, _) => result -- evaluates to 1

case (1, 2) of (_, result) => result -- evaluates to 2
```

La coincidencia de patrones también funciona para tuplas con más de dos componentes.

```
case (1, 2, 3) of (result, _, _) => result -- evaluates to 1

case (1, 2, 3) of (_, result, _) => result -- evaluates to 2

case (1, 2, 3) of (_, _, result) => result -- evaluates to 3
```

Haskell no proporciona funciones estándar como `fst` o `snd` para tuplas con más de dos componentes. La biblioteca de `tuple` en Hackage proporciona dichas funciones en el módulo `Data.Tuple.Select` .

Aplique una función binaria a una tupla (uncurrying)

Utilice la función `uncurry` (de `Prelude` o `Data.Tuple`) para convertir una función binaria en una función en tuplas.

```
uncurry (+) (1, 2) -- computes 3

uncurry map (negate, [1, 2, 3]) -- computes [-1, -2, -3]

uncurry uncurry ((+), (1, 2)) -- computes 3

map (uncurry (+)) [(1, 2), (3, 4), (5, 6)] -- computes [3, 7, 11]

uncurry (curry f) -- computes the same as f
```

Aplicar una función de tupla a dos argumentos (currying)

Utilice la función de `curry` (de `Prelude` o `Data.Tuple`) para convertir una función que lleva tuplas a una función que toma dos argumentos.

```
curry fst 1 2 -- computes 1

curry snd 1 2 -- computes 2

curry (uncurry f) -- computes the same as f

import Data.Tuple (swap)
curry swap 1 2 -- computes (2, 1)
```

Intercambiar pares de componentes

Use `swap` (de `Data.Tuple`) para intercambiar los componentes de un par.

```
import Data.Tuple (swap)
swap (1, 2) -- evaluates to (2, 1)
```

O utilice la coincidencia de patrones.

```
case (1, 2) of (x, y) => (y, x) -- evaluates to (2, 1)
```

Rigidez de emparejar una tupla.

El patrón `(p1, p2)` es estricto en el constructor de tuplas más externo, lo que puede llevar a un comportamiento inesperado de rigor . Por ejemplo, la siguiente expresión diverge (usando `Data.Function.fix`):

```
fix $ \(x, y) -> (1, 2)
```

ya que la coincidencia en `(x, y)` es estricta en el constructor de tuplas. Sin embargo, la siguiente expresión, que utiliza un patrón irrefutable , se evalúa como `(1, 2)` como se esperaba:

```
fix $ \ ~(x, y) -> (1, 2)
```

Lea Tuplas (pares, triples, ...) en línea: https://riptutorial.com/es/haskell/topic/5342/tuplas--pares--triples------

Capítulo 75: Usando GHCi

Observaciones

GHCI es el REPL interactivo que viene incluido con GHC.

Examples

Iniciando GHCi

Escriba `ghci` en un intérprete de comandos de shell para iniciar GHCI.

```
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Prelude>
```

Cambiar el indicador predeterminado de GHCi

De forma predeterminada, el indicador de GHCI muestra todos los módulos que ha cargado en su sesión interactiva. Si tienes muchos módulos cargados esto puede alargarse:

```
Prelude Data.List Control.Monad> -- etc
```

El comando `:set prompt` cambia el prompt para esta sesión interactiva.

```
Prelude Data.List Control.Monad> :set prompt "foo> "
foo>
```

Para cambiar la solicitud de forma permanente, agregue `:set prompt "foo> "` al archivo de configuración de GHCi .

El archivo de configuración de GHCi.

GHCi usa un archivo de configuración en `~/.ghci` . Un archivo de configuración consta de una secuencia de comandos que GHCi ejecutará en el inicio.

```
$ echo ":set prompt \"foo> \"" > ~/.ghci
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from ~/.ghci
foo>
```

Cargando un archivo

El `:l` o `:load` comando de `:load` comprueba y carga un archivo.

```
$ echo "f = putStrLn \"example\"" > example.hs
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
ghci> :l example.hs
[1 of 1] Compiling Main               ( example.hs, interpreted )
Ok, modules loaded: Main.
ghci> f
example
```

Dejar de fumar GHCi

Puede abandonar GHCi simplemente con `:q` o `:quit`

```
ghci> :q
Leaving GHCi.

ghci> :quit
Leaving GHCi.
```

Alternativamente, el acceso directo `CTRL` + `D` (`Cmd` + `D` para OSX) tiene el mismo efecto que `:q` .

Recargando un archivo ya cargado

Si ha cargado un archivo en GHCi (por ejemplo, utilizando `:l filename.hs`) y ha cambiado el archivo en un editor fuera de GHCi, debe volver a cargar el archivo con `:r` o `:reload` para hacer uso de los cambios, por lo tanto no es necesario escribir de nuevo el nombre del archivo.

```
ghci> :r
OK, modules loaded: Main.

ghci> :reload
OK, modules loaded: Main.
```

Puntos de ruptura con GHCi

GHCi admite puntos de interrupción de estilo imperativo fuera de la caja con código interpretado (código que se ha `:loaded`).

Con el siguiente programa:

```
-- mySum.hs
doSum n = do
  putStrLn ("Counting to " ++ (show n))
  let v = sum [1..n]
  putStrLn ("sum to " ++ (show n) ++ " = " ++ (show v))
```

cargado en GHCi:

```
Prelude> :load mySum.hs
[1 of 1] Compiling Main             ( mySum.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Ahora podemos establecer puntos de interrupción utilizando números de línea:

```
*Main> :break 2
Breakpoint 0 activated at mySum.hs:2:3-39
```

y GHCi se detendrá en la línea correspondiente cuando ejecutemos la función:

```
*Main> doSum 12
Stopped at mySum.hs:2:3-39
_result :: IO () = _
n :: Integer = 12
[mySum.hs:2:3-39] *Main>
```

Puede ser confuso dónde estamos en el programa, por lo que podemos usar `:list` para aclarar:

```
[mySum.hs:2:3-39] *Main> :list
1  doSum n = do
2    putStrLn ("Counting to " ++ (show n))   -- GHCi will emphasise this line, as that's where
we've stopped
3    let v = sum [1..n]
```

Podemos imprimir variables, y continuar la ejecución también:

```
[mySum.hs:2:3-39] *Main> n
12
:continue
Counting to 12
sum to 12 = 78
*Main>
```

Declaraciones multilínea

La instrucción `:{` inicia el *modo multilínea* y `:}` finaliza. En el modo multilínea, GHCi interpretará las nuevas líneas como puntos y coma, no como el final de una instrucción.

```
ghci> :{
ghci| myFoldr f z [] = z
ghci| myFoldr f z (y:ys) = f y (myFoldr f z ys)
ghci| :}
ghci> :t myFoldr
myFoldr :: (a -> b -> b) -> b -> [a] -> b
```

Lea Usando GHCi en línea: https://riptutorial.com/es/haskell/topic/3407/usando-ghci

Capítulo 76: Vectores

Observaciones

> [Data.Vector] hace hincapié en un rendimiento muy alto a través de la fusión de bucles, al tiempo que conserva una interfaz rica. Los principales tipos de datos son matrices en caja y sin caja, y las matrices pueden ser inmutables (puras) o mutables. Las matrices pueden contener elementos almacenables, adecuados para pasar a C y desde ella, y puede convertir entre los tipos de matriz. Las matrices están indexadas por valores de Int no negativos.

El Wiki de Haskell tiene estas recomendaciones :

> En general:
>
> - Los usuarios finales deben usar Data.Vector.Unboxed para la mayoría de los casos
> - Si necesita almacenar estructuras más complejas, use Data.Vector
> - Si necesita pasar a C, use Data.Vector.Storable
>
> Para escritores de bibliotecas;
>
> - Use la interfaz genérica para asegurarse de que su biblioteca sea lo más flexible posible: Data.Vector.Generic

Examples

El módulo Data.Vector

El módulo Data.Vector proporcionado por el vector es una biblioteca de alto rendimiento para trabajar con arreglos.

Una vez que haya importado `Data.Vector` , es fácil comenzar a usar un `Vector` :

```
Prelude> import Data.Vector
Prelude Data.Vector> let a = fromList [2,3,4]

Prelude Data.Vector> a
fromList [2,3,4] :: Data.Vector.Vector

Prelude Data.Vector> :t a
a :: Vector Integer
```

Incluso puedes tener una matriz multidimensional:

```
Prelude Data.Vector> let x = fromList [ fromList [1 .. x] | x <- [1..10] ]

Prelude Data.Vector> :t x
```

```
x :: Vector (Vector Integer)
```

Filtrando un vector

Filtrar elementos impares:

```
Prelude Data.Vector> Data.Vector.filter odd y
fromList [1,3,5,7,9,11] :: Data.Vector.Vector
```

Mapeo (`map`) y reducción (` fold`) de un vector

Los vectores se pueden `map` y `fold'd`, filtrar `'d and` comprimir:

```
Prelude Data.Vector> Data.Vector.map (^2) y
fromList [0,1,4,9,16,25,36,49,64,81,100,121] :: Data.Vector.Vector
```

Reducir a un solo valor:

```
Prelude Data.Vector> Data.Vector.foldl (+) 0 y
66
```

Trabajando en múltiples vectores

Zip dos matrices en una matriz de pares:

```
Prelude Data.Vector> Data.Vector.zip y y
fromList [(0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10),(11,11)] ::
Data.Vector.Vector
```

Lea Vectores en línea: https://riptutorial.com/es/haskell/topic/4738/vectores

Capítulo 77: XML

Introducción

Codificación y decodificación de documentos XML.

Examples

Codificación de un registro utilizando la biblioteca `xml`

```
{-# LANGUAGE RecordWildCards #-}
import Text.XML.Light

data Package = Package
  { pOrderNo  :: String
  , pOrderPos :: String
  , pBarcode  :: String
  , pNumber   :: String
  }

-- | Create XML from a Package
instance Node Package where
  node qn Package {..} =
    node qn
      [ unode "package_number" pNumber
      , unode "package_barcode" pBarcode
      , unode "order_number" pOrderNo
      , unode "order_position" pOrderPos
      ]
```

Lea XML en línea: https://riptutorial.com/es/haskell/topic/9264/xml

Capítulo 78: zipWithM

Introducción

`zipWithM` es para `zipWith` como `mapM` es para `map` : te permite combinar dos listas usando una función monádica.

Desde el módulo `Control.Monad`

Sintaxis

- zipWithM :: Applicative m => (a -> b -> mc) -> [a] -> [b] -> m [c]

Examples

Cálculos de precios de venta.

Supongamos que desea ver si un determinado conjunto de precios de venta tiene sentido para una tienda.

Los artículos originalmente cuestan $ 5, por lo que no desea aceptar la venta si el precio de venta es menor para cualquiera de ellos, pero sí quiere saber cuál es el precio nuevo.

Calcular un precio es fácil: usted calcula el precio de venta y no devuelve `Nothing` si no obtiene una ganancia:

```
calculateOne :: Double -> Double -> Maybe Double
calculateOne price percent = let newPrice = price*(percent/100)
                             in if newPrice < 5 then Nothing else Just newPrice
```

Para calcularlo para toda la venta, `zipWithM` hace realmente simple:

```
calculateAllPrices :: [Double] -> [Double] -> Maybe [Double]
calculateAllPrices prices percents = zipWithM calculateOne prices percents
```

Esto devolverá `Nothing` si alguno de los precios de venta está por debajo de $ 5.

Lea zipWithM en línea: https://riptutorial.com/es/haskell/topic/9685/zipwithm

Creditos

S. No	Capítulos	Contributors
1	Empezando con Haskell Language	4444, Adam Wagner, alejosocorro, Amitay Stern, arseniiv, baxbaxwalanuksiwe, Benjamin Hodgson, Benjamin Kovach, Burkhard, Carsten, colinro, Community, Daniel Jour, Echo Nolan, erisco, gdziadkiewicz, HBu, J Atkin, Jan Hrcek, Jules, Kwarrtz, leftaroundabout, M. Barbieri, mb21, mnoronha, Mr Tsjolder, ocharles, pouya, R B, Ryan Hilbert, Sebastian Graf, Shoe, Stephen Leppik, Steve Trout, Tim E. Lord, Turion, user239558, Will Ness, zbw, λlεx
2	Agujeros mecanografiados	Cactus, leftaroundabout, user2407038, Will Ness
3	Analizando HTML con lentes etiquetadas y lentes	Kostiantyn Rybnikov, λlεx
4	Apilar	4444, curbyourdogma, Dair, Janos Potecki
5	Aritmética	ocramz
6	Atravesable	Benjamin Hodgson, ErikR, J. Abrahamson
7	Attoparsec	λlεx
8	Bases de datos	λlεx
9	Bifunctor	Benjamin Hodgson, liminalisht
10	Cábala	tlo
11	Categoria teoria	arrowd, Benjamin Hodgson, Benjamin Kovach, J. Abrahamson, Mario Román, mmlab, user2407038
12	Clases de tipo	arjanen, Benjamin Hodgson, Billy Brown, Cactus, Dair, gdziadkiewicz, J. Abrahamson, Kwarrtz, leftaroundabout, mnoronha, RamenChef, Will Ness, zeronone, λlεx
13	Clasificación de los algoritmos	Brian Min, pouya, Romildo, Shoe, Vektorweg, Will Ness
14	Composición de funciones	arseniiv, Will Ness

15	Comprobación rápida	Benjamin Hodgson, gdziadkiewicz, Matthew Pickering, Steve Trout
16	Concurrencia	Janos Potecki, λlεx
17	Contenedores - Data.Map	Cactus, Itbot, Will Ness, λlεx
18	Creación de tipos de datos personalizados	Janos Potecki, Kapol
19	Data.Aeson - JSON en Haskell	Chris Stryczynski, Janos Potecki, Matthew Pickering, rob, xuh
20	Data.Text	Benjamin Hodgson, dkasak, Janos Potecki, jkeuhlen, mnoronha , unhammer
21	Declaraciones de fijeza	Alec, Will Ness
22	Desarrollo web	arrowd, λlεx
23	Esquemas de recursion	arseniiv, Benjamin Hodgson
24	Explotación florestal	λlεx
25	Extensiones de lenguaje GHC comunes	Antal Spector-Zabusky, Bartek Banachewicz, Benjamin Hodgson, Benjamin Kovach, Cactus, charlag, Doomjunky, Janos Potecki, John F. Miller, K48, Kwarrtz, leftaroundabout, Mads Buch, Matthew Pickering, mkovacs, mniip, phadej, Yosh, λlεx
26	Fecha y hora	mnoronha, Will Ness, λlεx
27	Flechas	artcorpse, leftaroundabout
28	Función de sintaxis de llamada.	Zoey Hewll
29	Funciones de orden superior	Community, Doruk, Matthew Pickering, mnoronha, Will Ness, λlεx
30	Functor	Benjamin Hodgson, Delapouite, Janos Potecki, liminalisht, mathk, mnoronha, Will Ness, λlεx
31	Functor Aplicativo	Benjamin Hodgson, Kritzefitz, mnoronha, Will Ness, λlεx
32	GHCJS	Mikkel

33	Google Protocol Buffers	Janos Potecki, λlεx
34	Gráficos con brillo	Wysaard, Zoey Hewll
35	Gtk3	bleakgadfly
36	Interfaz de función extranjera	arrowd, bleakgadfly, crockeea
37	IO	3442, Benjamin Hodgson, David Grayson, J. Abrahamson, Jan Hrcek, John F. Miller, leftaroundabout, mnoronha, Sarah, user2407038, Will Ness, λlεx
38	Lector / Lector	Chris Stryczynski
39	Lente	Bartek Banachewicz, bennofs, chamini2, dfordivam, dsign, Hjulle, J. Abrahamson, John F. Miller, Kwarrtz, Matthew Pickering, λlεx
40	Lista de Comprensiones	Cactus, Kwarrtz, mnoronha, Will Ness
41	Literales sobrecargados	Adam Wagner, Carsten, Kapol, leftaroundabout, pdexter
42	Liza	Benjamin Hodgson, erisco, Firas Moalla, Janos Potecki, Kwarrtz , Lynn, Matthew Pickering, Matthew Walton, Mirzhan Irkegulov, mnoronha, Tim E. Lord, user2407038, Will Ness, Yosh, λlεx
43	Los funtores comunes como base de los comonads cofree.	leftaroundabout
44	Mejoramiento	λlεx
45	Módulos	Benjamin Hodgson, Kapol, Will Ness, λlεx
46	Mónada del estado	Apoorv Ingle, Kritzefitz, Luis Casillas
47	Mónadas	Alec, Benjamin Hodgson, Cactus, fgb, Kapol, Kwarrtz, Lynn, Mario Román, Matthew Pickering, Will Ness, λlεx
48	Mónadas comunes como mónadas libres.	leftaroundabout
49	Mónadas Libres	Benjamin Hodgson, Cactus, J. Abrahamson, pyon, sid-kap

50	Monoide	Benjamin Hodgson, Kwarrtz, mnoronha, Will Ness
51	Operadores de infijo	leftaroundabout, mnoronha
52	Papel	xuh
53	Paralelismo	λlεx
54	Plantilla Haskell & QuasiQuotes	user2407038
55	Plátano reactivo	arrowd, Dair, Undreren
56	Plegable	Benjamin Hodgson, Cactus, Dair, David Grayson, J. Abrahamson, Jan Hrcek, mnoronha
57	Polimorfismo de rango arbitrario con RankNTypes	ocharles
58	Probando con Tasty	tlo
59	Profesor	zbw
60	Proxies	Benjamin Hodgson
61	Reglas de reescritura (GHC)	Cactus
62	Rigor	Benjamin Hodgson, Benjamin Kovach, Cactus, user2407038, Will Ness
63	Sintaxis de grabación	Cactus, Janos Potecki, John F. Miller, Mario, Matthew Pickering , Will Ness, λlεx
64	Sintaxis en funciones	Delapouite, James, Janos Potecki, Will Ness, λlεx
65	Solicitud parcial	arseniiv, Benjamin Hodgson, CiscoIPPhone, Dair, Matthew Pickering, Will Ness
66	Streaming IO	λlεx
67	Tipo algebra	Mario Román
68	Tipo de solicitud	Luka Horvat, λlεx
69	Tipo Familias	leftaroundabout, mniip, xuh
70	Tipos de datos algebraicos generalizados	mniip

71	Tipos fantasma	Benjamin Hodgson, Christof Schramm
72	Transformadores de mónada	Damian Nadales
73	Tubería	4444, Benjamin Hodgson, Stephane Rolland, λlεx
74	Tuplas (pares, triples, ...)	Cactus, mnoronha, Toxaris, λlεx
75	Usando GHCi	Benjamin Hodgson, James, Janos Potecki, mnoronha, RamenChef, wizzup, λlεx
76	Vectores	Benjamin Hodgson, λlεx
77	XML	Mikkel
78	zipWithM	zbw