TP4 - ALP - 2020

Introducción

Hola! Este README es un documento complementario al PDF de la consigna del TP. La mayor parte de esta guía es idéntica a la del TP1, pero es recomendable leer con atención la sección **Estructura del código**.

Stack

Para este TP vamos a usar **Stack**, una herramienta sencilla para desarrollar poryectos en Haskell. Stack tiene muchas utilidades, pero ahora nos vamos a concentrar sus funciones básicas.

Antes que nada, puede que tengas que instalarlo. En 1 hay guías de instalación para distintas plataformas.

Stack se encarga de instalar la versión correcta de GHC, instalar los paquetes necesarios y compilar el proyecto. Para las primeras dos, basta con abrir una terminal en el directorio `TP4` y ejecutar:

```
stack setup
```

Esto puede demorar un rato porque se encarga de descargar e instalar la verisón correcta de GHC. Este comando solo se debería tener que ejecutar una única vez. Al terminar esto, está todo listo para compilar el proyecto, que se hace con:

```
stack build
```

Este es el comando que van a tener que usar para compilar el proyecto cada vez que lo modifiquen.

Estructura del código

La estructura del proyecto es la siguiente:

```
.
├── app
│   └── Main.hs
├── src
│   ├── AST.hs
│   ├── Eval1.hs
│   ├── Eval2.hs
│   ├── Eval3.hs
│   ├── Monads.hs
│   ├── PPLis.hs
│   └── Parser.hs
├── Ejemplos
│   ├── div.lis
│   ├── error1.lis
│   ├── error2.lis
│   └── sqrt.lis
├── test
├── README.md
├── Setup.hs
├── TP4.cabal
├── package.yaml
├── stack.yaml
└── stack.yaml.lock
```

IMPORTANTE: Solo deberían tener que modificar archivos de los directorios `src` y, si quieren experimentar, `Ejemplos`.

- En el directorio `app` se define el módulo `Main`, que implementa el ejecutable final.
- En el directorio `src` se encuentran los módulos sobre los que van a trabajar:
  - `AST` define los tipos de expresiones y comandos presentados en la consigna junto a algunos tipos auxiliares.
  - `Eval1` tiene el esqueleto para el primer evaluador.
  - `Eval2` tiene el esqueleto para el segundo evaluador.
  - `Eval3` tiene el esqueleto para el tercer evaluador.
  - `Monads` define las clases de mónadas que proveen las operacione necesarias para implementar los evaluadores.
  - `Parser` esta vez el parser les viene gratis, generado autmáticamente por Happy.
  - `PPLis` tiene el Pretty Printer del lenguaje LIS. Este sirve para imprimir los programas de una manera más legible que haciendo `show` sobre el AST (y viene de regalo).
- En el directorio `Ejemplos` hay algunos -*shock*- ejemplos de programas LIS.
- El resto de los archivos son de configuración del proyecto.

IMPORTANTE: Por favor, no cambiar los nombres de los módulos, tipos, constructores, funciones, etc. Ante cualquier duda consulte a su docente de cabecera.

¿Cómo ejecutarlo?

Una vez compilado el proyecto, se puede correr el ejecutable definido en `app/Main.hs` sobre un archivo `.lis` haciendo:

```
stack exec TP4-exe -- PATH_TO_SOURCE [-OPT]
```

Las opciones disponibles son:

- `-p` : Imprimir el programa de entrada.
- `-a` : Mostrar el AST del programa de entrada.
- `-e N_EVALUADOR` : Elegir evaluador 1, 2 o 3 (1 por defecto).
- `-h` : Imprimir ayuda.

Por ejemplo, para imprimir el programa `div.lis` del directorio `Ejemplos` , ejecutar:

```
stack exec TP4-exe -- Ejemplos/div.lis -p
```

Para correrlo con el evaluador de `Eval2.hs` (que van a tener que definir ustedes):

```
stack exec TP4-exe -- Ejemplos/div.lis -e 2
```

Inicialmente ambos comandos van a generar errores, pero a medida de que vayan realizando el TP van a poder utilizarlos (`-p` y `-a` dependen de `Parser` , mientras que `-e n` depende de `Eval` n).

¿Y GHCi?

Si quieren usar `GHCi` para probar alguna de las funciones que definieron, pueden iniciar una sesión del intérprete con todos los módulos del directorio `src` ya cargados haciendo:

```
stack ghci
```

Comentarios importantes (y no tanto)

Ahora les voy a hacer algunos comentarios sobre algunas herramientas de Haskell/GHC que no usamos en EDyA 2. No son muy difíciles, pero viene bien tener una referencia.

GADTs

La más importante es el uso de GADTs. Estos son tipos de datos algebraicos generalizados, que permiten que sus constructores instancien el tipo que parametriza el ADT. Pueden ver un explicaciones extensas en 2, 3 y 4, pero probablemente sea suficiente el siguiente ejemplo.

Supongamos que queremos representar en Haskell el siguiente lenguaje de expresiones enteras y booleanas:

```
Term ::=   const Int
         | true
         | false
         | suc Term
         | isZero Term
```

Es decir, un término puede ser una constante entera, un valor booleano, la aplicación de `suc` a un término o la aplicación de `isZero` a un término.

En Haskell esto se podría representar como:

```haskell
data Term a = Const Int
            | VTrue
            | VFalse
            | Suc (Term a)
            | IsZero (Term a)
```

Sin embargo, nos gustaría que solo se puedan representar los términos que tienen sentido. En concreto, queremos que no se puedan expresar términos como `Suc (IsZero VFalse)`, ya que nuestra idea es que `suc` solo se pueda aplicar sobre expresiones enteras y que `isZero` solo se aplique a expresiones booleanas.

Para esto mismo sirven los GADTs, que requieren la extensión `GADTs` de GHC (ya está incluída en el proyecto). Esta extensión nos da una nueva sintaxis para definir tipos de datos, en la que se le asigna a cada constructor su tipo explícitamente.
Por ejemplo, el tipo `Term` se puede definir con esta sintaxis de la siguiente manera:

```haskell
data Term a where
  Const  :: Int -> Term a
  VTrue  :: Term a
  VFalse :: Term a
  Suc    :: Term a -> Term a
  IsZero :: Term a -> Term a
```

Esta nueva declaración es equivalente a la anterior, pero ahora nos permite hacer algo interesante con el tipo `a` que parametriza a `Term`. Lo que vamos a hacer es instanciar `a` a distintos tipos,

dependiendo del constructor:

```
data Term a where
  Const  :: Int -> Term Int
  VTrue  :: Term Bool
  VFalse :: Term Bool
  Suc    :: Term Int -> Term Int
  IsZero :: Term Int -> Term Bool
```

Ahora, esta declaración deja en claro sobre qué términos opera cada constructor. De esta manera, términos como `Suc (IsZero VFalse)` no son representables por este tipo de datos. Esto es muy bueno, porque nos asegura que todo valor de tipo `Term Int` va a ser una expresión entera bien formada, y lo mismo para `Term Bool`.

Otra ventaja de esta última declaración, es que nos permite definir un evaluador como:

```
eval :: Term a -> a
eval (Const n) = n
eval VTrue     = True
...
```

En el que el tipo resultado de la función depende del término que se está evaluando.

El tipo de datos que representa las expresiones en LIS `Exp a` se define siguiendo esta misma idea.

Comentario: Para derivar instancias (como `Eq` y `Show`) de un GADTs hace falta otra extensión llamada `StandaloneDeriving` (también incluída en el proyecto). Esta permite escribir la cláusula `deriving` separada de la definición del ADT, como puede verse en `src/AST.hs`.

Map

Para representar el estado en el módulo `Eval1` se utiliza el tipo `Map` de `Data.Map.Strict`. Un mapa de tipo `Map k v` representa una asociación de claves de tipo `k` a valores de tipo `v`. En la documentación de este módulo, disponible en 5, se describen todos las funciones que permiten manejarlos.

Tuplas estrictas

También el el módulo `Eval1`, se utiliza el tipo `Pair a b`. Este es parte del módulo `Data.Strict.Tuple`. Un elemento de `Pair a b` es esencialmente un par de valores `x` de tipo `a` e `y` de tipo `b`. Estos se pueden construir como `(x :!: y)`.

La diferencia con el valor `(x, y)` de tipo `(a, b)` es que en `(x :!: y)` los valores `x` e `y` son evaluados de forma estricta (recordar que Haskell tiene evaluación peresoza por defecto). Esto va a venir bien para entender mejor el orden en el que se evalúan las expresiones.

No deberían necesitar mucho más sobre este tipo, pero, como siempre, su documentación está en 6.

Pattern Synonyms

La extensión `PatternSynonyms` (ya incluída en el proyecto) permite definir sinónimos para los constructores de un tipo de datos. Esto también posibilita abstraer ciertas construcciones útiles que no tienen una representación real en el tipo de datos.

Por ejemplo, el cómando `if b then c` de la sintaxis concreta no tiene una representación directa en la sintaxis abstracta, ya que esta puede representarse como `if b then c else skip`. Esto se puede expresar en Haskell de la siguiente manera, presente en el módulo `AST`:

```
pattern IfThen :: Exp Bool -> Comm -> Comm
pattern IfThen b c = IfThenElse b c Skip
```

Con esto, se puede usar el sinónimo `IfThen b c` tanto para hacer pattern-matching como para contruir valores.

Para más detalles de esta extensión, ver 7.

Referencias

[1] - https://docs.haskellstack.org/en/stable/README/#how-to-install

[2] - https://downloads.haskell.org/~ghc/6.6/docs/html/users_guide/gadt.html

[3] - https://en.wikipedia.org/wiki/Generalized_algebraic_data_type

[4] - http://dev.stephendiehl.com/hask/#gadts

[5] - https://hackage.haskell.org/package/containers-0.6.3.1/docs/Data-Map-Strict.html

[6] - http://hackage.haskell.org/package/strict-0.4/docs/Data-Strict-Tuple.html

[7] - https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/pattern_synonyms.html