25-2-2008

1. Registro de los cambios (II)

- Se propone un método para registrar los cambios en la topología resultado de aplicar un mecanismo de transición. En primer lugar, se declara el tipo enumerado *TipoCambio*, que identifica a cada una de las reglas.

```
data TipoCambio = Operador1 | Operador2 | ...
```

- Se ha pensado que la manera más sencilla de registrar los cambios es por medio de una tupla que contenga: el identificador de la regla aplicada, el nodo inicial de la red a partir del cual se aplica, el número de nodos sobre los que se aplica, y el número de nodos resultantes después de aplicarla.

```
tipo Cambios = (TipoCambio, Int, Int, Int)
```

- Como ya se comentó anteriormente, los cambios se guardan como una lista en el árbol de soluciones. La función *terminales* devuelve las hojas del árbol (posibles soluciones), que incluyen el escenario final y una lista con los cambios necesarios para llegar hasta él. Se ha definido el tipo de datos *Solucion* para simplificar la declaración de estas funciones:

```
type Solucion = (Escenario, [Cambios])
terminales :: Arbol → [Solucion]
```

- Para calcular el coste de una solución particular, es necesario primero una función que calcule el coste de un cambio. Suponemos que cada cambio es independiente del resto. Una de las ventajas de usar un lenguaje funcional, es que en lugar de limitarnos a un criterio concreto, podemos definir varias de estas funciones y pasarlas como parámetro. Definimos el siguiente tipo de datos:

```
type Coste = Cambios → Double
```

- Con esta función, calcular el coste de la solución es tan sencillo como calcular el coste de cada cambio por separado y sumar todos los valores:

```
coste_solucion :: Coste → Solucion → Double
coste_solucion f (e, cambios) = foldr (+) 0 (map f cambios)
```

- Por último, quedaría ordenar las soluciones por orden creciente de coste:

```
ordenar :: Coste → [Solucion] → [(Solucion,Double)]
ordenar f xs = sortBy orden $ zip xs (map (coste_sol f) xs)
    where
    orden (_,c1) (_,c2) = compare c1 c2
```

2. Reorganización del código

- Al empezar a implementar la base de reglas entera me he dado cuenta de que es muy extensa, por lo que convenía reorganizarla. Estudiando más a fondo las reglas, he visto que se dividen en cuatro categorías:
  - Creación de zonas.
  - Operadores de conexión.
  - Canonización.
  - Mecanismos de transición.
- Lo primero que he hecho ha sido separar la implementación de cada tipo de regla en un módulo diferente. Estos módulos son *Zonas.hs*, *Conexiones.hs*, *Canonizacion.hs* y *Mecanismos.hs* en el directorio *Reglas*.
- He visto que los tres primeros tipos de reglas se aplican secuencialmente, con el objetivo de simplificar el escenario lo más posible antes de empezar a probar los mecanismos de transición. Por tanto, se pueden aplicar con un función aparte, y no considerarlas en la generación del árbol, lo que reduce considerablemente el tamaño de éste.
- Creo que la implementación del árbol ya está terminada. Quizás haya que modificar algo, pero serán cambios menores. Por tanto, he decidido limpiar la interfaz del módulo. Ahora exporta el tipo *Arbol*, el ADT que representa el árbol de soluciones, y un conjunto de funciones para crear y manipular el árbol. No se exportan los constructores. Las funciones son:
  - *generar_arbol*: crea el árbol a partir de la base de reglas y el escenario inicial.
  - *podar*: recorta aquellas ramas del árbol que no cumplan la condición pasada como parámetro.
  - *terminales*: devuelve una lista con las hojas del árbol.
- He creado el módulo *Pruebas.hs*, que se encarga de ejecutar las pruebas unitarias del resto de módulos. Conforme se me vayan ocurriendo más pruebas las iré añadiendo aquí.

3. QuickCheck

He estado probando el funcionamiento de QuickCheck. Parece bastante potente y sencillo de usar. A continuación comento las primeras impresiones:

- El funcionamiento básico de QuickCheck consiste en definir *propiedades*. En esencia, una propiedad es una función que toma unos valores y verifica si se cumplen determinadas propiedades de una función. Por ejemplo, la siguiente función comprueba que dos reglas de creación de zonas son conmutativas:

```
prop_conmutativa xs = (zonas1 . zonas2 $ xs) == (zonas2 . zonas1 $ xs)
```

- Las propiedades se invocan llamando a la función *quickCheck*, que toma como parámetro la propiedad a verificar. En principio, genera 100 casos de prueba aleatorios. Si la propiedad se cumple para todos los casos, se considera que la prueba ha sido exitosa, mientras que si alguno de ellos falla, se muestra el caso que ha fallado. Por ejemplo, para verificar el ejemplo anterior, escribiríamos lo siguiente:

```
> quickCheck prop_conmutativa
OK, passed 100 tests.
```

- Nota: como necesita generar valores aleatorios, la función *quickCheck* funciona en la mónada *IO*.

- También existe la función *verboseCheck*, que muestra cada caso antes de probarlo.
- QuickCheck necesita un par de funciones auxiliares para poder generar los valores aleatorios. Estas funciones están recogidas en la clase *Arbitrary*. Se proporcionan instancias para los tipos de datos básicos (cadenas, enteros, listas, etc.), pero si se quieren usar tipos definidos por el usuario hay que implementarlos manualmente. De momento, se ha implementado la clase para los tipos *Router* y *Conexion*.
- QuickCheck también proporciona una biblioteca de combinadores que permiten construir propiedades más complejas. Por ejemplo, el operador de implicación *a* ==>*b* verifica que si la propiedad se cumple para *a*, también tiene que cumplirse para *b*.
- Quizás el aspecto más complicado sea determinar que propiedades deben cumplir las funciones. De momento, he definido tres propiedades triviales:
  - La aplicación de una regla de simplificación siempre reduce el número de nodos de la red (o por lo menos lo mantiene igual).
  - Dadas dos reglas de simplificación, la aplicación de ambas es conmutativa.
  - Tras la aplicación de las reglas de creación de zonas no puede ocurrir que haya dos zonas $\{Z_4, Z_6, Z_d\}$ adyacentes del mismo tipo.
- Existe la posibilidad de modificar los parámetros de QuickCheck, haciendo que pruebe un mayor número de casos por ejemplo.

Parece que QuickCheck es una herramienta bastante útil. Aunque no cubre todos los tipos de pruebas que podemos hacer, y desde luego no es un sustituto de una batería de pruebas al estilo tradicional, si que es un buen complemento. Al probar casos aleatorios es bastante probable encontrar alguno que no hubieramos tenido en cuenta al definir la batería de pruebas. Además, QuickCheck ayuda a documentar el código a modo de especificación funcional.

4. HaRP

También he estado mirando HaRP. En principio, parece ser muy potente, en la documentación aparecen ejemplos de todo tipo de expresiones regulares complejas sobre listas muy interesantes. El problema es que no hay una versión estable. He tenido que actualizar todo el GHC para poder compilarlo. Aún no he podido probarlo, en cuanto haga decidiré si es útil usarlo o no.