16-1-2008

1. Registro de los cambios

- Debemos decidir una manera de registrar la secuencia de cambios que se aplican para llegar del escenario inicial al final. Interesa que la representación sea compacta y eficiente, evitando redundancias. De momento supondremos que hay definido un tipo de datos llamado CAMBIOS.
- En primer lugar, hay que modificar las reglas para que devuelvan los cambios aplicados:

```
data Cambios = ...
type Regla = Red → Maybe (Red, Cambios)
```

- También es necesario disponer de una función que recorra el árbol y nos devuelva las posibles soluciones (nodo terminal) junto con la lista de cambios necesarios para llegar a ellas:

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

- La implementación de la función anterior depende de cómo registremos estos cambios en el árbol de soluciones. Este es quizás el paso más complicado, ya que se presentan muchas alternativas.
  - La primera opción sería guardar en cada nodo los cambios que se han aplicado para llegar del nodo padre al actual. Para obtener las soluciones habría que recorrer el árbol acumulando los cambios en una lista. La estructura resultante sería:

    ```
    data Arbol = Arbol Red Cambios [Arbol]
    ```

  - La estructura anterior tiene un defecto, y es que el nodo raíz no tiene padre, y por tanto tampoco cambios. Esto nos obligaría a introducir algún tipo de cambio nulo. Para evitar este problema podemos registrar los cambios que se producen para llegar del nodo actual a los nodos hijos:

    ```
    data Arbol = Arbol Red [(Cambios, Arbol)]
    ```

  - La tercera opción es guardar en cada nodo la lista completa de cambios. La longitud de esta lista sería proporcional al nivel del nodo en el árbol, siendo 0 para el nodo raíz. Los cambios se registrarían en el orden en el que se producen, añadiéndolos al final de la lista.

    ```
    data Arbol = Arbol Red [Cambios] [Arbol]
    ```

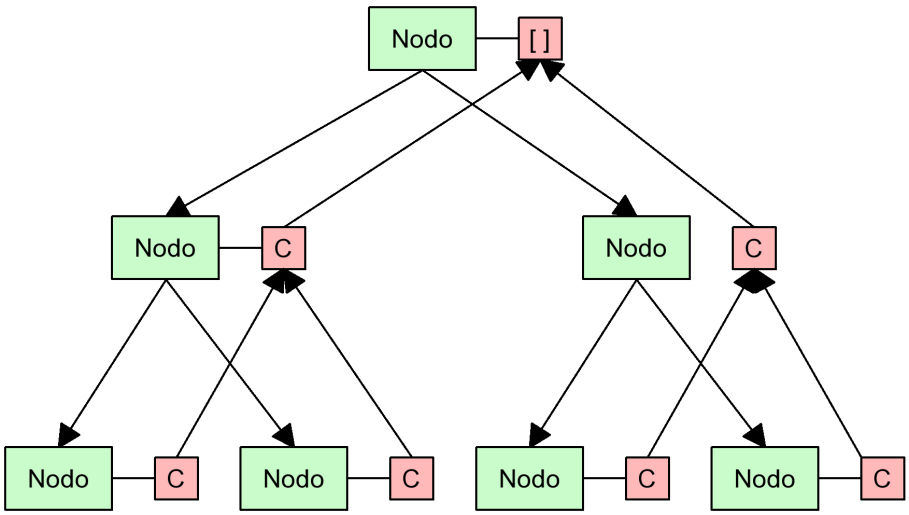
  - La opción anterior tiene un gran defecto, y es que almacena mucha información redundante. Por ejemplo, si un nodo tiene una lista de cambios $[c_1, c_2, c_3 \dots c_n]$, sus hijos tendrían una lista $[c_1, c_2, c_3 \dots c_n, c_{n+1}]$. Es decir, para cada nodo se repiten n elementos, siendo n la profundidad del nodo en el árbol. Por tanto, el espacio desperdiciado crece exponencialmente. Sin embargo, podemos aprovechar una característica de los lenguajes funcionales. Éstos suelen definir la listas de manera recursiva, lo que permite que varias listas compartan la misma cola. Si guardamos la lista de cambios en orden inverso, es decir, $[c_n \dots c_3, c_2, c_1]$, los nodos hijos tendrían una lista $[c_{n+1}, c_n \dots c_3, c_2, c_1]$ y podrían compartir la lista de cambios del padre. La estructura resultante es la misma que en el caso anterior, sólo cambia la forma en la que se registran los cambios.

    ```
    data Arbol = Arbol Red [Cambios] [Arbol]
    ```

- Cada una de las soluciones anteriores tiene una serie de ventajas y defectos. Las dos primeras son las más compactas, y también las que más facilitan la generación del árbol, pero a costa de complicar el recorrido. Las dos últimas son el caso opuesto. Quizás la opción más equilibrada sea la última, por lo que es la que elegiremos.

2. Implementación

- Elegimos por tanto la cuarta opción. La estructura resultante tiene el siguiente aspecto:



- Hay que reescribir las funciones de generación del árbol para que anoten los cambios.

```
-- Intenta aplicar todas las reglas a una red
probar_reglas :: [Regla] → Red → [(Red, Cambios)]
probar_reglas reglas red = filtrar $ rmap reglas red

-- Añade un nodo al principio de varias listas
insertar :: (Router, Conexion) → [(Red,Cambios)] → [(Red, Cambios)]
insertar e xs = [(e:elems, cambios) | (elems, cambios) ← xs]

-- Expande un nodo del árbol, probando todas las reglas posibles
expandir_nodo :: [Regla] → Red → [(Red, Cambios)]
expandir_nodo _ [] = []
expandir_nodo reglas red@(r:rs) = (probar_reglas reglas red)
                  ++ (insertar r $ expandir_nodo reglas rs)

-- Genera el árbol a partir de la red inicial
generar_arbol :: [Regla] → Red → Arbol
generar_arbol reglas red = generar' reglas red []
    where
    generar' reglas red cambios = Nodo red cambios hijos
        where
        hijos = [generar' reglas r (c:cambios)
                | (r,c) ← expandir_nodo reglas red]
```

- A continuación definimos una función que, dado un árbol, busca todas las soluciones posibles (es decir, los nodos terminales). Como ya se ha dicho anteriormente, hay que invertir la lista de cambios.

```
terminales :: Arbol → [Solucion]
terminales (Nodo red cambios []) = [(red, reverse cambios)]
terminales (Nodo _ _ hijos) = concatMap buscar_soluciones hijos
```

3. Trabajos futuros

A partir de las funciones que se han implementado queda casi terminado el núcleo del programa. En principio, sólo quedaría implementar la poda, el criterio de selección de soluciones, y la reconstrucción del camino seguido.

- Por el momento se puede prescindir de la poda. He definido una función PODAR que simplemente devuelve el árbol sin modificar. Más adelante, cuando esté más avanzado el proyecto, podremos entrar en detalles e implementarla.
- El criterio de selección sería una función que, dada una posible solución, devuelva un valor booleano indicando si es correcta o no. Sólo habría que filtrar la lista devuelta por TERMINALES para tener la lista de soluciones.
- El paso más complejo será reconstruir la secuencia de pasos a partir de la lista de cambios. La implementación de esta función va a estar estrechamente ligada a la manera en la que definamos el tipo de datos CAMBIOS. Queda por decidir esta representación.
- Quedaría por implementar la base de reglas, y definir un banco de pruebas inicial.

4. Herramientas y bibliotecas externas

En esta sección voy a comentar una serie de ideas que se me han ocurrido y que podrían ayudar en el desarrollo del proyecto.

- Convendría ir documentando y verificando el código correctamente, para no encontrarnos con sorpresas desagradables más adelante.
- Para la documentación la solución estándar suele ser Haddock, una herramienta similar a Doxygen pero para Haskell. Habría que añadir anotaciones al código que permitirán más adelante generar documentación en HTML.
- Para la verificación la herramienta más usada suele ser QuickCheck. A partir de una lista de propiedades que debe cumplir cada función genera un banco de pruebas, pensado para probar todas las condiciones posibles. Sería interesante pensar qué pruebas pueden diseñarse para este programa.
- Para cargar los ficheros de grafos se pueden usar tres opciones. La más directa, y probablemente la que mayor rendimiento proporcione, sea escribir el cargador manualmente. También es la más compleja, por lo que se descarta. Otra opción es usar Parsec, una biblioteca de combinadores que permite escribir analizadores dentro del propio Haskell. Es muy flexible y potente, y además tiene la ventaja de que no requiere herramientas externas, pero tiene dos desventajas: al combinar el análisis léxico y sintáctico en una sola pasada se puede complicar la gramática, y la compilación del código resultante es extremadamente lenta. La tercera opción es usar Alex y Happy, dos herramientas similares a Lex y Yacc.
- Como ya dije, hay algunas reglas que son de longitud variable. Una opción es usar HaRP, un preprocesador que permite utilizar expresiones regulares en patrones con listas. HaRP lee un fichero en Haskell con anotaciones especiales, y lo traduce a Haskell puro. Por desgracia, usa algunas extensiones propias del GHC, que no todos los compiladores soportan. Conviene por tanto evaluar la complejidad de estas reglas y decidir si es mejor usar HaRP o implementarlas a mano.
- Por último, y ya más a largo plazo, habría que elegir qué biblioteca voy a usar para implementar la interfaz gráfica. Hay muchas opciones, pero la mayoría son experimentales. Las únicas que parecen más maduras son wxHaskell y gtk2hs. Sólo he conseguido compilar la última, así que la elección parece clara. (Nota: desde la última vez que lo probé han aparecido bibliotecas nuevas)