7-12-2007

1. Representación de los escenarios

- La primera pregunta que surge es: ¿cómo representamos los escenarios, y más concretamente, la red que conecta los nodos finales?
- Parece claro que para los nodos, routers, y aplicaciones finales que forman la red utilizamos un tipo enumerado.

```
data Aplicacion = A4 | A6 | Ad
data Nodo = N4 | N6 | Nd | Nmap
data Router = R4 | R6 | Rd | R4t ...
```

- ¿Qué ocurre con los operadores de conexión? Si sólo hubiera uno, podríamos usar una simple lista, pero como hay varios tenemos que recurrir a una estructura más compleja.
- Si analizamos en profundidad el problema, se puede ver que los escenarios son, por así decirlo, expresiones matemáticas que hay que simplificar por medio de una serie de reglas. La manera habitual de representar expresiones matemáticas es por medio de un árbol, pero como todos los operadores tienen la misma precedencia y asociatividad, basta con una estructura lineal.
- Podemos considerar por tanto tres opciones:
  - Definir un tipo enumerado para los routers y otro para los operadores, y representar la red como una lista de tuplas *(Router,Operador)*, donde el campo *Operador* indica el mecanismo de conexión usado entre ese router y el siguiente nodo de la red. Los tipos de datos resultantes serían:

    ```
    data Operador = Directa | OpD | Op4t | OpDt ...
    type Red = [(Router,Operador)]
    ```

  - La representación anterior tiene un problema, y es que permite definir redes vacías. Si queremos evitar esto, podemos definir nuestro propio tipo de datos lineal que garantice que las redes tengan siempre al menos un elemento:

    ```
    data Operador = Directa | OpD | Op4t | OpDt ...
    data Red = Rama Router Red
             | Hoja Router
    ```

  - La tercera opción es similar a la anterior, pero en lugar de representar los operadores como un tipo enumerado, lo hacemos por medio de los constructores de la propia estructura de datos. Esto tiene la ventaja de que las reglas se pueden representar de una manera más compacta y natural, pero tiene la desventaja de que en algunas reglas aparecen operadores variables, lo que obliga a añadir un caso por cada posible operador. El código resultante sería:

    ```
    data Red = Directa Router Red
             | OpD Router Red
             | Op4t Router Red
             ...
             | Final Nodo
    ```

- De momento no está clara cual es la mejor opción. Se ha elegido la primera, ya que parece que es la que permite escribir las reglas de la manera más simple.

2. Representación de las reglas

- Puesto que estamos trabajando con un lenguaje funcional, lo más sencillo es representar cada regla con una función.

- Necesitamos una manera de indicar que una regla no es aplicable. Para ello, podemos usar el tipo MAYBE: si una regla tiene éxito, devuelve JUST X, y si no se puede aplicar, devuelve NOTHING. Definimos un tipo para las reglas:

```
type Regla = Red → Maybe Red
```

- Aquí surgen dos dudas:
  - ¿Cómo representamos las reglas de longitud variables? La mayoría de las reglas son de longitud fija, y está claro que para esos casos lo más sencillo es utilizar patrones. Pero en algunas reglas el número de nodos puede variar. Hay una extensión que permite usar expresiones regulares en los patrones, que quizás podría servir. La otra opción es implementar estas reglas a mano, aunque sea más difícil.
  - ¿Cómo anotamos los cambios efectuados por cada regla? Necesitamos conocer los cambios para poder reconstruir el camino seguido una vez que alcancemos una solución válida.
- Las funciones que representan las reglas tendrían la siguiente forma:

```
regla1 :: Regla
regla1 ((zi,Directa) : (Rd,Directa) : zj) = Just $ (zi,OpD) : zj
regla1 _ = Nothing
```

- Definimos una lista que contenga a todas las funciones anteriores, para poder referenciar rápidamente todas las reglas:

```
reglas :: [Reglas]
reglas = [regla1, regla2, ... reglaN]
```

3. Generación del árbol de búsqueda

- Una vez que sabemos como representar los escenarios y hemos definido las reglas, podemos generar el árbol de búsqueda.
- Existen múltiples maneras de representar el árbol. Por ejemplo, podríamos definir constructores distintos para las ramas y las hojas, o entre hojas que son solución y hojas que no lo son, pero creemos que no es necesario. Puesto que el árbol es N-ario, los hijos de un nodo son una lista. Un nodo es terminal si la lista está vacía.

```
data Arbol a = Arbol a [Arbol a]
```

- Definimos una función que aplica una lista de funciones (reglas) a un único valor (escenario). Puesto que esta función es la inversa de *map*, se le ha denominado *rmap*.

```
rmap :: [a→b] → a → [b]
rmap fs x = [f x | f <- fs]
```

- Filtramos para eliminar los valores NOTHING.

```
filtrar :: [Maybe Red] → [Red]
filtrar [] = []
filtrar (Just x : xs) = x : filtrar xs
filtrar (Nothing : xs) = filtrar xs
```

- Combinamos estas dos funciones en una tercera que prueba todas las reglas posibles para un escenario concreto:

```
probar_reglas :: [Regla] → Red → [Red]
probar_reglas reglas red = filtrar $ rmap reglas red
```

- A continuación, creamos una función que obtenga todos los resultados posibles de aplicar las reglas a un nodo del árbol. Hay que tener en cuenta que las reglas pueden aplicarse no sólo al principio de la red, sino también en el medio.

```
expandir_nodo :: [Regla] → Red → [Red]
expandir_nodo _ [] = []
expandir_nodo reglas red@(r:rs) = (probar_reglas reglas red) ++ (map (r:) $ expandir_no
```

- Una vez que tenemos definidas estas funciones, generar el árbol es trivial. Aquí tenemos la ventaja de que al usar un lenguaje perezoso podemos definir el árbol entero sin miedo a quedarnos sin memoria.

```
generar_arbol :: [Regla] → Red → Arbol
generar_arbol reglas red = Arbol red hijos
        where
        hijos = map (generar_arbol reglas) $ expandir_nodo reglas red
```

- Una vez generado el árbol, queda procesarlo. Aquí surgen muchas dudas:
  - El árbol puede ser muy profundo, por lo que necesitamos podarlo lo antes posible. ¿Qué criterios de poda hay?
  - ¿Cómo se determina que una solución (nodo terminal) es correcta? Supongo que sólo sirven las redes de un nodo.
  - ¿Cómo reconstruimos la secuencia de pasos a través del árbol?