Universidad del Valle

Facultad de ingeniería

Ingeniería en sistemas

Cristian David Pacheco Torres
2227437
Juan Sebastián Molina Cuéllar
2224491

October 12, 2023

Taller 3: Reconocimiento de patrones.

Contents

1 Maniobras en trenes.

1.1 Aplicar movimiento.

1.1.1 Informe de uso del reconocimiento de patrones.

Reconocimiento de patrones		
Función	¿Se utilizó pattern Matching?	¿Razón?
AplicarMovimiento	Sí	La función utiliza *pattern matching* para analizar el tipo de Movimiento y determinar la lógica de transición de estados. Dependiendo del movimiento y su magnitud, se ajustan las posiciones de los elementos en las tres sub-listas del estado, garantizando que el estado resultante refleje adecuadamente el movimiento aplicado.

Table 1: Tabla para la función *AplicarMovimiento*

```scala
def aplicarMovimiento(state:Estado, mov:Movimiento):
   Estado = {
    mov match {
        case Uno(n) if(n > 0) => (state._1.dropRight(n
           ),  state._1.takeRight(n) ::: state._2,
           state._3)
        case Dos(n) if(n > 0) => (state._1.dropRight(n
           ), state._2, state._1.takeRight(n) :::
           state._3)
        case Uno(n) if(n < 0) => (state._1 ::: state.
           _2.take(-n),  state._2.drop(-n), state._3)
        case Dos(n) if(n < 0) => (state._1 ::: state.
           _3.take(-n), state._2, state._3.drop(-n))
        case _  => (state._1, state._2, state._3)
    }
}
```

Listing 1: Código en Scala para la funcion aplicarMovimiento

La función **aplicarMovimiento** toma dos argumentos: un estado (*state*) y un movimiento (*mov*), y retorna un nuevo estado. Utiliza el reconocimiento de patrones para analizar el tipo y las propiedades del *mov* proporcionado y, dependiendo de este, aplica diferentes lógicas para modificar el state y generar un nuevo estado. Los estados y los movimientos están representados como tuplas y casos, respectivamente, y la función se asegura de manejar diferentes escenarios posibles (movimientos positivos/negativos y de dos tipos diferentes

Uno y Dos) para actualizar el estado de manera adecuada. En caso de que el movimiento no coincida con ninguno de los patrones definidos, el estado se mantiene sin cambios.

1.1.2 Informe de Corrección.

Argumentación sobre la corrección:
$\forall\ s \in Estado :\ Estado\ \{Tren1, Tren2, Tren3\}\ \wedge\ Tren\ \{List[Vagon]\}\ \wedge\ Vagon\{Any\}$
$\forall\ m \in Movimiento : Movimiento\ \{Uno(Int), Dos(Int)\} \wedge Int\ \{\mathbb{Z}\}$

Caso base:

- $if\ (m = (Uno(Int) \vee Dos(Int)) \wedge Int = 0) \rightarrow case_{\_}\ Estado(Tren1, Tren2, Tren3)$

Caso posibles:

- $if\ (m = (Uno(Int)) \wedge Int > 0) \rightarrow$
  $Estado(Tren1.dropRight(Int), Tren1.takeRight(Int)\ :::\ Tren2, Tren3)$
- Si $(m = (\text{Dos}(n)) \wedge n > 0) \rightarrow$
  $\text{Estado}(Tren1.\text{dropRight}(n), Tren2, Tren1.\text{takeRight}(n)\ :::\ Tren3)$
- Si $(m = (\text{Uno}(n)) \wedge n < 0) \rightarrow$
  $\text{Estado}(Tren1\ :::\ Tren2.\text{take}(-n), Tren2.\text{drop}(-n), Tren3)$
- Si $(m = (\text{Dos}(n)) \wedge n < 0) \rightarrow$
  $\text{Estado}(Tren1\ :::\ Tren3.\text{take}(-n), Tren2, Tren3.\text{drop}(-n))$

Caso prueba:

Si tenemos:
$valstate = (List(a, b, c), List(d), List(e, f)) \wedge valmov = Uno(2)$

Remplazando :
$aplicarMovimiento(state, mov) =$
$aplicarMovimiento((List(a, b, c), List(d), List(e, f)), Uno(2))$

Por el caso "$Uno(Int) \wedge Int > 0$" :
$= (List(a, b, c).dropRight(2), List(a, b, c).takeRight(2)\ :::\ List(d), List(e, f))$
$= (List(a), List(b, c, d), List(e, f))$

Casos de prueba:

```
val e6 = (List('a', 'b', 'c','d'), List(), List())
val e7 = aplicarMovimiento(e6, Uno(2)) //Esperado: (List('a', 'b'), List('c','d'), List())
val e8 = aplicarMovimiento(e7, Dos(1)) //Esperado: (List('a'), List('c','d'), List('b'))
val e9 = aplicarMovimiento(e8, Uno(-1)) //Esperado: (List('a','c'), List('d'), List('b'))
val e10 = aplicarMovimiento(e9, Dos(-1)) //Esperado: (List('a','c', 'b'), List('d'), List())
val e11 = aplicarMovimiento(e10, Uno(1)) //Esperado: (List('a','c'), List('b','d'), List())
val e12 = aplicarMovimiento(e11, Dos(2)) //Esperado: (List(), List('b','d'), List('a','c'))
```

Listing 2: Casos de prueba para la funcion aplicarMovimiento

1.1.3 Conclusión.

La utilización de un modelo de sustitución y casos de prueba para validar la corrección del algoritmo *aplicarMovimiento* fue esencial para asegurar su funcionalidad en diversos escenarios y movimientos. El modelo de sustitución permite descomponer y entender la lógica detrás de cada movimiento y transición de estado, mientras que los casos de prueba aseguran que el algoritmo actúa de manera esperada en situaciones específicas y límites. Juntos, proporcionan un marco robusto para verificar la integridad, la robustez y la fiabilidad del algoritmo, garantizando que los movimientos y estados resultantes sean coherentes y precisos.

1.2 Aplicar movimientos

1.2.1 Informe de uso del reconocimiento de patrones.

Reconocimiento de patrones		
Función	¿Se utilizó pattern Matching?	¿Razón?
AplicarMovimientos	Sí	La función emplea *pattern matching* para descomponer recursivamente la lista de movimientos (*maniobras*) en cabeza y cola. A través de esta descomposición, aplica secuencialmente cada movimiento al estado actual, acumulando y retornando una lista de todos los estados intermedios generados durante la aplicación de la secuencia de movimientos.

Table 2: Tabla para la función *AplicarMovimientos*

```
def aplicarMovimientos(state:Estado, movs:Maniobra):
    List[Estado] = {
  def helper(state:Estado, movs:Maniobra, acc:List[
      Estado]): List[Estado] = movs match {
      case Nil => acc
      case x::xs => {
          val newState = aplicarMovimiento(state, x)
          helper(newState, xs, acc :+ newState)
      }
  }
  helper(state, movs, List())
}
```

Listing 3: Código en Scala para la funcion aplicarMovimientos

La función **aplicarMovimientos** toma un *estado* y una lista de *movimientos*, y devuelve una lista de *estados* resultantes de aplicar esos *movimientos* secuencialmente. Utiliza una función auxiliar *helper* para realizar esta tarea de manera recursiva, aplicando cada movimiento al *estado* actual y acumulando los *estados* intermedios en una lista, que se devuelve cuando todos los *movimientos* han sido aplicados.

1.2.2 Informe de Corrección.

$s\forall \in Estado : Estado\{Tren1, Tren2, Tren3\} \wedge Tren\{List[Vagon]\} \wedge Vagon\{Any\}$
$m\forall \in Maniobra : Maniobra\{List[Movimiento]\} \wedge Movimiento\{Uno(Int), Dos(Int)\} \wedge Int\{\mathbb{Z}\}$

Caso inicial:

- $S_0 = (s_{s_0}, List(mov_0, .., mov_n), List())$

$$mov \in Movimiento \wedge i, e, c, n \in \mathbb{Z}^+$$

- $S = (S_{s_k}, List(mov_k, .., mov_n), List(List(Estado_0), ..., List(Estado_k)))$
- $S_f = (s_{s_n}, List(), List(Estado_0), ..., List(Estado_n))$
- $Inv(S_{s_k}, m_k, e_k) \equiv e_k \subset \{s_1 \cup s_2 \cup s_3\} \ \cup \ aplicarMovimiento(m_k)$
- $Transformar\ ((s_1, s_2, s_3), m_k, e_k) = (\{s_1 \cup s_2 \cup s_3\} \cup aplicarMovimiento(m_k), m_{k1}, e_k + aplicarMovimiento(e_k))$ donde (s_1, s_2, s_3) define el estado de la estacion, m_k, y e_k es la lista de movimientos restantes y estados aplicados hasta la iteracion K, respectivamente.

Hipótesis Inductiva (HI):

Asumimos que para un estado arbitrario s_k y una lista de movimientos m_k de longitud $n - k$, la función `aplicarMovimientos` produce una lista de estados e_k tal que cada estado en e_k es el resultado de aplicar los movimientos en m_k al estado inicial s_0.

$$HI(k) \equiv aplicarMovimientos(s_k, m_k) = \{e_k\} \cup \{s_k\}$$

Paso Inductivo:

Queremos mostrar que si la hipótesis inductiva es verdadera para k, entonces también lo es para $k + 1$.

$$HI(k+1) \equiv aplicarMovimientos(s_{k+1}, m_{k+1}) = \{e_{k+1}\} \cup \{s_{k+1}\}$$

Considerando un movimiento m_k y el estado s_{k+1} que es el resultado de aplicar mov_k a s_k, queremos mostrar que:

$$aplicarMovimientos(s_k, m_k) = \{e_{k+1}\} \cup \{s_{k+1}\}$$

Por la definición de la función `aplicarMovimientos`, sabemos que:

$$aplicarMovimientos(s_k \cup \{e_k\}, m_{k+1}) = aplicarMovimientos(s_{k+1}, m_{k+1}) \cup \{e_{k+1}\}$$

Por la hipótesis inductiva, sabemos que:

$$aplicarMovimientos(s_k, m_k) = e_k$$

Por lo tanto, podemos concluir que:

$$aplicarMovimientos(s_k, m_k) = e_{k+1} \cup \{s_{k+1}\}$$

Esto completa el paso inductivo y, por lo tanto, la corrección de la función `aplicarMovimientos` ha sido demostrada por inducción. Y asá cada vez la lista se va volviendo más pequeña y va actualizando el estado con los movimientos que va consumiendo hasta que la lista de movimientos se vacia.

Casos de prueba:

```
val e13 = (List('a', 'b', 'c'), List('d'), List('e', '
   f'))
aplicarMovimientos(e13, List(Uno(2), Dos(1)))
aplicarMovimientos(e13, List(Uno(-1), Dos(2), Uno(1)))
aplicarMovimientos(e13, List(Dos(1), Uno(-1), Dos(-1))
   )
val e14 = (List('x', 'y'), List('z', 'a', 'b'), List('
   c'))
aplicarMovimientos(e14, List(Dos(2), Uno(1)))
aplicarMovimientos(e14, List(Uno(-1), Dos(-2), Uno(2))
   )
```

Listing 4: Para ver los valores esperados por favor referirse al archivo de pruebas.sc. Debido a que los resultados son tan extensos que dañan la estructura del documento.

1.2.3 Conclusión.

Su correctitud la garantiza la inducción estructural, debido a que se utiliza la recursión para llamarse a si misma. La inducción estructural nos muestra que de una lista de movmimientos al aplicarlos consecutivamente, nos llevará a nuevos estdos que representaran los movimientos de un tren y sus diferentes trayectorias, hasta acabar con la posibilidad de moverse.

En cuanto al uso de pattern matching en algoritmos, este enfoque permite una estructuración clara y legible del código, facilitando la identificación y manejo de diversos casos y subcasos de manera ordenada y explícita.En particular en este problema, determinar el movimiento siguiente a realizar y comprobar cuándo ha llegado al final de la lista de movimientos dada.

1.3 Definir maniobras

1.3.1 Informe de uso del reconocimiento de patrones.

Reconocimiento de patrones		
Función	¿Se utilizó pattern Matching?	¿Razón?
$DefinirManiobra$	Sí	La función utiliza *pattern matching* para descomponer el estado deseado y el estado actual en sus componentes respectivos, permitiendo la comparación elemento por elemento entre ellos. A través de una serie de reglas y condiciones, se generan movimientos que transforman el estado inicial en el estado deseado, considerando las diferencias entre los elementos y sus posiciones en las respectivas $sub-listas$.

Table 3: Tabla para la función $DefinirManiobra$

```
 def definirManiobra(initialState:Tren,wantedState:Tren):Maniobra = {
   def moveHelper(state: Estado, wantedState: Tren, movs: List[Movimiento]): List[Movimiento] = {
     wantedState match {
         case Nil => movs
         case x::xs if(!state._1.isEmpty && x == state._1.head) => {
           moveHelper((state._1.tail,state._2,state._3), xs, movs)
         }
         case x::xs => {
             ...ver codigo en package.scala....
         }
         case _ => movs
     }
   }
   moveHelper((initialState, List(), List()), wantedState, List())
}
```

Listing 5: Código en Scala para la funcion aplicarMovimientos

La función **definirManiobra** busca determinar una secuencia de movimientos (*maniobra*) que transforme un estado inicial de un tren (*initialState*) en un estado deseado (*wantedState*). Utiliza una función auxiliar *moveHelper* que, mediante el uso de pattern matching, evalúa y compara los elementos de los estados actual y deseado, generando los movimientos necesarios para alinear cada componente del tren en la posición deseada, y acumulando estos movimientos en una lista que se devuelve como resultado.

1.3.2 Informe de Corrección.

Sea k $\in N$, $0 \leq k \leq n$, un entero que indica el número actual de maniobras en el trayecto principal T_p, $S_0 = < b_1, \ldots,\ b_i,\ \ldots, b_{n-1}, b_n >$ una secuencia que define el estado inicial en T_p; $S_1 =< c_1, c_2,\ \ldots\ , b_{n-1}, c_n >$, $0 \leq j \leq n-1$ elementos, una secuencia que define el estado de un trayectto secundario T_2 en el paso k, $S_2 =< e_1, e_2,\ \ldots\ , e_{n-1}, e_n >$, $0 \leq j \leq n-1$ elementos, una secuencia que define el estado de un trayectto secundario T_2 en el paso k; una función l que determina el número de elementos de la secuencia s de entrada; y $P_{k-n}^{k-n}(S_k)$ una funcion de permutacion de $k-n$ en $k-n$ elementos sobre los elementos de secuencia S en el paso k de la maniobra.
Por premisa se tiene que $S_0[i] = Sd[j]$, para $0 \leq i \leq n$ y $0 \leq j \leq n$
Se quiere demostar que $\exists\ S_n =< a_1, a_2,\ \ldots\ , a_{n-1}, a_n >\ \mid S_n[i] = Sd[i]$ para $0 \leq i \leq n$

Por tanto, se tiene

- Un estado $s = (S_k, S_1, S_2, S_d, m)$ donde
  - S_k representa la secuencia en la iteracion k.
  - S_1 el estado sobre el trayecto T_1.
  - S_2 el estado sobre el trayecto T_2.
  - Sd el estado deseado.
  - m la lista de moviemientos (maniobras) hasta el paso k.
- El estado inicial $S_0 = (S_0, [\ \ ], [\ \ ], S_d, [\ \ ])$.
- $S_f = (S_n, [\ \ ], [\ \ ],\ S_d,\ m)$. donde $S_n[i] = S_d[i]$ en $0 \leq i \leq n$.

- $Inv(S_k, S_1, S_2, S_d, m) \equiv \exists\ p := P_{n-k}^{n-k}(Si\ k = 0 \to S_0 \vee S_1 + S_2)$
  $\mid p[n-k:n] = S_d[n-k:n] \ \ \wedge\ l(m) \leq (n-1) + \ \ \ldots\ \ + (n-k+1),$
  $\wedge\ \ S_d[i] = T(S_k)[i],\ \ \forall i \mid 0 \leq i \leq n-k \wedge 0 \leq k \leq n$

- $transformar(S_k, S_1, S_2, S_d, m) =$

$$T(S_k, S_1, S_2, S_d, m) = \begin{cases} ((S_k[k:n],\ S_1,\ S_2),\ S_d[k:n],\ m), \\ \qquad\qquad\qquad \text{Si } k \neq 0 \wedge k \neq n \wedge\ S_k[0] = S_d[0] \\ (S_k,\ [b_k] + S_1,\ S_2),\ S_d,\ m + (Uno(1)), \\ \qquad\qquad\qquad \text{Si } S_k \neq [\ ] \wedge S_K = (S_k,\ S_1,\ [\ \ ]) \wedge l(S_k) = 1 \\ (S_k, [\ \ ], [\ \ ]), S_d,\ [\ \ ] + (Uno(-n)), \\ \qquad\qquad\qquad \text{Si } k \neq 0 \wedge k \neq n \wedge\ S_k[0] \neq S_d[0] \\ (S_k,\ S_1,\ [b_k, \ldots, b_n]),\ S_d,\ m + (Dos(k-n)), \\ \qquad\qquad\qquad \text{Si } S_k \neq [\ ] \wedge S_K = (S_k,\ S_1,\ [\ \ ]) \wedge l(S_1) = 1 \\ (S_k,\ S_1[l(S_1) - 1 : l(S_1)],\ S_2),\ S_d,\ m + (Uno(-l(S_1) - 1)), \\ \qquad\qquad\qquad \text{Si } S_k \neq [\ ] \wedge S_K = (S_k,\ S_1,\ [\ \ ]) \\ (S_k + [c_1],\ [\ ],\ S_2),\ S_d,\ m + (Uno(-1)), \\ \qquad\qquad\qquad \text{Si } S_k \neq [\ ] \wedge S_K = (S_k,\ S_1,\ S_2) \wedge l(S_1) = 1 \\ (S_k + [e_1],\ S_1,\ [\ ]),\ S_d,\ m + (Dos(-1)), \\ \qquad\qquad\qquad \text{Si } S_k \neq [\ ] \wedge S_K = (S_k,\ S_1,\ S_2) \wedge l(S_2) = 1 \\ ([\ ] + [b_k : b_{n-1}] + S_1,\ S_2),\ S_d,\ m + (Uno(l(S_k))), \\ \qquad\qquad\qquad \text{Si } S_k \neq [\ ] \wedge S_K = (S_k,\ [\ ],\ S_2) \wedge l(S_2) = 1 \\ (S_k + [e_k : e_{n-1}],\ [\ ],\ [\ e_1]),\ S_d,\ m + (Dos(-l(S_2) - 1)), \\ \qquad\qquad\qquad \text{Si } S_k \neq [\ ] \wedge S_K = (S_k,\ [\ ],\ S_2) \\ ((S_K, S_1, S_2),\ S_d,\ m), \qquad Si\ S_k = (\_, \_, \_) \end{cases}$$

Ahora se procede a demostrar la correctitud de los enunciados anteriores. Para el estado inicial S_0, se tiene:

$Inv(S_0) \to Inv(S_0,\ [\ ],\ [\ ], S_d,\ [\ ])$, para $k = 0$ iteraciones se cumple que existe una permutacion $p = P_n^n(S_0)$, tal que lleva S_0 a S_d, pues $S_0 \subseteq S_d \wedge S_d \subseteq S_0$.
Por otro lado, $S_k = S_0$, el estado no se ha modificado mediante la transformacion T. Ádemas, $l(m) = 0 \leq k - n$ se cumple.

$$(S_k \neq S_f \wedge Inv(S_k)) \to Inv(T(S_{k+1})) \equiv$$

$(\exists\ p := P_{n-k}^{n-k}(S_1 + S_2)\ |\ p[n-k : n] = S_d[n-k : n] \quad \wedge\ l(m) \leq (n-1) +\ \ldots\ + (n-k+1), \wedge\ S_d[i] = T(S_k)[i], para\ i\ |\ 0 \leq i \leq n-k) \rightarrow (\exists\ p := P_{n-k+1}^{n-k+1}([S_1 + S_2] - [a \in S_1 + S_2])\ |\ p[n-k+1 : n] = S_d[n-k+1 : n] \wedge\ l(m) \leq (n-1) +\ \ldots\ + (n-k+2), \wedge\ S_d[i] = T(S_{k+1})[i], para\ i\ |\ 0 \leq i \leq n-k+1)$

Y, por ultimo, se tiene para S_f

$$Inv(S_f) \rightarrow S_n = S_d \equiv$$

$$(\exists\ p := P_0^0([\ ])\ |\ p[\ ] = [\ ] \quad \wedge\ l(m) \leq \tfrac{n(n-1)}{2} \wedge\ S_d[i] = S_n[i],\ para\ 0 \leq i \leq n)$$

$$\equiv$$

$$true\ \wedge\ true\ \wedge\ true \equiv true$$

Casos de prueba:

```
definirManiobra(List('a', 'b', 'c'), List('c', 'b', 'a'))
definirManiobra(List('w', 'x', 'y', 'z', 'a'), List('a', 'z', 'y', 'x', 'w'))
definirManiobra(List('m', 'n'), List('n', 'm'))
definirManiobra(List('g', 'h', 'i', 'j', 'k', 'l'), List('l', 'k', 'j', 'i', 'h', 'g'))
definirManiobra(List('t', 'u', 'v'), List('v', 'u', 't'))
```

Listing 6: Para ver los valores esperados por favor referirse al archivo de pruebas.sc. Debido a que los resultados son tan extensos que dañan la estructura del documento.

1.3.3 Conclusión.

La combinación de una demostración por inducción estructural y casos de prueba meticulosos proporciona una base sólida para afirmar la correctitud del algoritmo $definirManiobra$. La inducción estructural, que se ha aplicado previamente, asegura que el algoritmo funciona para todos los posibles estados y transiciones del sistema, validando su lógica inherente y estructura recursiva. Por otro lado, los casos de prueba, han sido diseñados para abordar diversas situaciones y escenarios, proporcionando una verificación empírica de la funcionalidad del algoritmo en contextos específicos. Juntos, estos dos enfoques complementan la validación teórica y práctica del algoritmo, asegurando que su diseño a través del *patternmatching* y ejecución son sólidos y confiables para definir los movimientos que deberían aplicarse para que un tren alcance un estado deseado.