

Testing Exhaustivo Acotado, Problemas y Posibles Soluciones

Valeria Bengolea

Diciembre 2013, FaMAF

Testing

- * Dada una pieza de software, el testing es la tarea de probarla con diferentes entradas.
- * Es una de las técnicas mas usadas para detectar errores en el software.
- * La obtención de entradas es una tarea esencial en testing.
- * Las entradas seleccionadas deberían “ejercitar” el software bajo diferentes escenarios.

Generación Automática de Casos de Test

- * La automatización de esta tarea es esencial en testing.
- * Casos de test que involucran tipos básicos pueden ser generados automáticamente de manera sencilla.
- * Esta tarea puede ser muy difícil cuando queremos testear rutinas paramétrizadas con **datos estructuralmente complejos** (grafos, árboles, etc).

Enfoques en la Generación Automática de Casos de Test

- * Generación Random
- * Generación Basada en Constraint Solving
 - * SAT solving, SMT solving, model checking, búsqueda, etc.
- * Generación exhaustiva acotada
 - * “Fuerza Bruta”, aunque en general requiere Constraint Solving

Generación Exhaustiva Acotada



- * Un enfoque para hacer testing en general, y para la generación de entradas
- * Consiste en generar TODAS las posibles entradas dentro de alguna cota dada.
- * Muy efectivo cuando se trata de rutinas paramétrizadas con estructuras complejas alojadas en memoria dinámica.

Testing Exhaustivo Acotado para estructuras complejas

- * Se necesita como entrada el rangos para los dominios, una descripción de las entradas válidas (precondición, Invariante de representación) y el código bajo prueba.
- * Una manera de generar todas las estructuras posibles en cierto rango es construir posibles candidatos y chequear si estos candidatos se corresponde con una estructura válida.
- * Para obtener todos los candidatos posibles en un rango de valores podríamos representar las estructuras como vectores candidatos.

Ejemplo de Invariante de Representación

```
public class SinglyLinkedList
{
    public Entry header;
    private int size = 0;
    ...
}
```

```
public class Entry {
    Object element;
    Entry next;
}
```

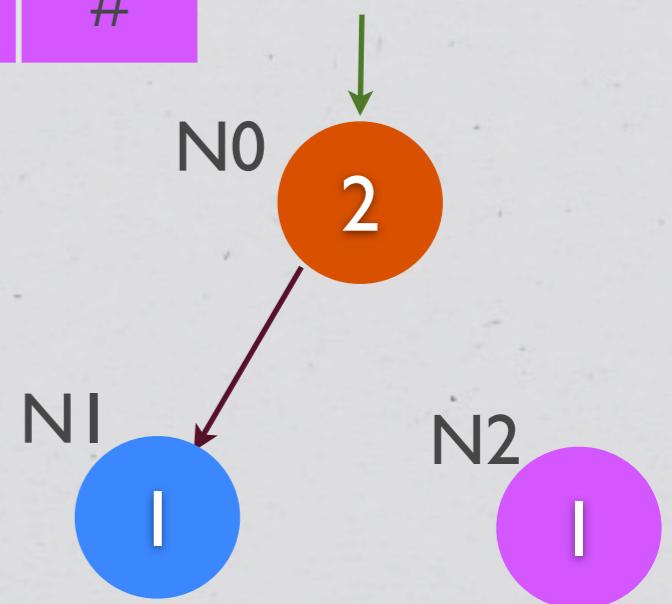
```
public boolean repOK() {
    if (header == null)
        return false;
    if (header.element != null)
        return false;
    Set<Entry> visited = new java.util.HashSet<Entry>();
    visited.add(header);
    Entry current = header;
    while (true) {
        Entry next = current.next;
        if (next == null)
            break;
        if (next.element == null)
            return false;
        if (!visited.add(next))
            return false;
        current = next;
    }
    if (visited.size() - 1 != size)
        return false;
    return true;
}
```

Testing Exhaustivo Acotado para estructuras complejas (Cont)

- * Supongamos que de deseamos generar todos los Arboles binarios de búsqueda (ABB) de hasta 3 nodos con claves entre 1 y 3, podríamos representarlos con el siguiente vector:

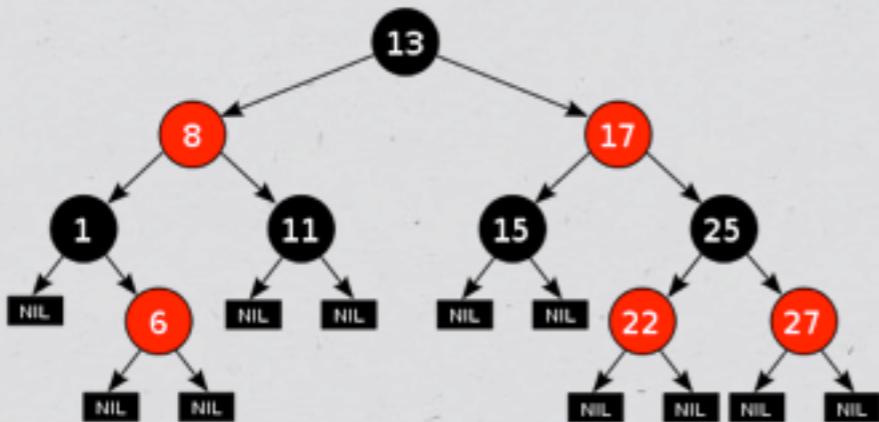
root	data	left	right	data	left	right	data	left	right
N0	2	NI	#	I	#	#	I	#	#

- * Para mirar todos las posibles estructuras es necesario probar todas las combinaciones (dentro de los rangos establecidos) de los campos de este vector.



Enumerar estructuras, que tan costoso puede ser?

Consideremos árboles rojos y negros (n nodos, m claves):



$n+1$ asignaciones a root

m^n asignaciones de claves a nodos

$(n+1)^n$ asignaciones al campo left

$(n+1)^n$ asignaciones al campo right

2^n asignaciones de colores a nodos

root	data	left	right	color	data	left	right	color	data	left	right	color
N0	2	NI	#	B	I	#	I	R	I	NI	#	R

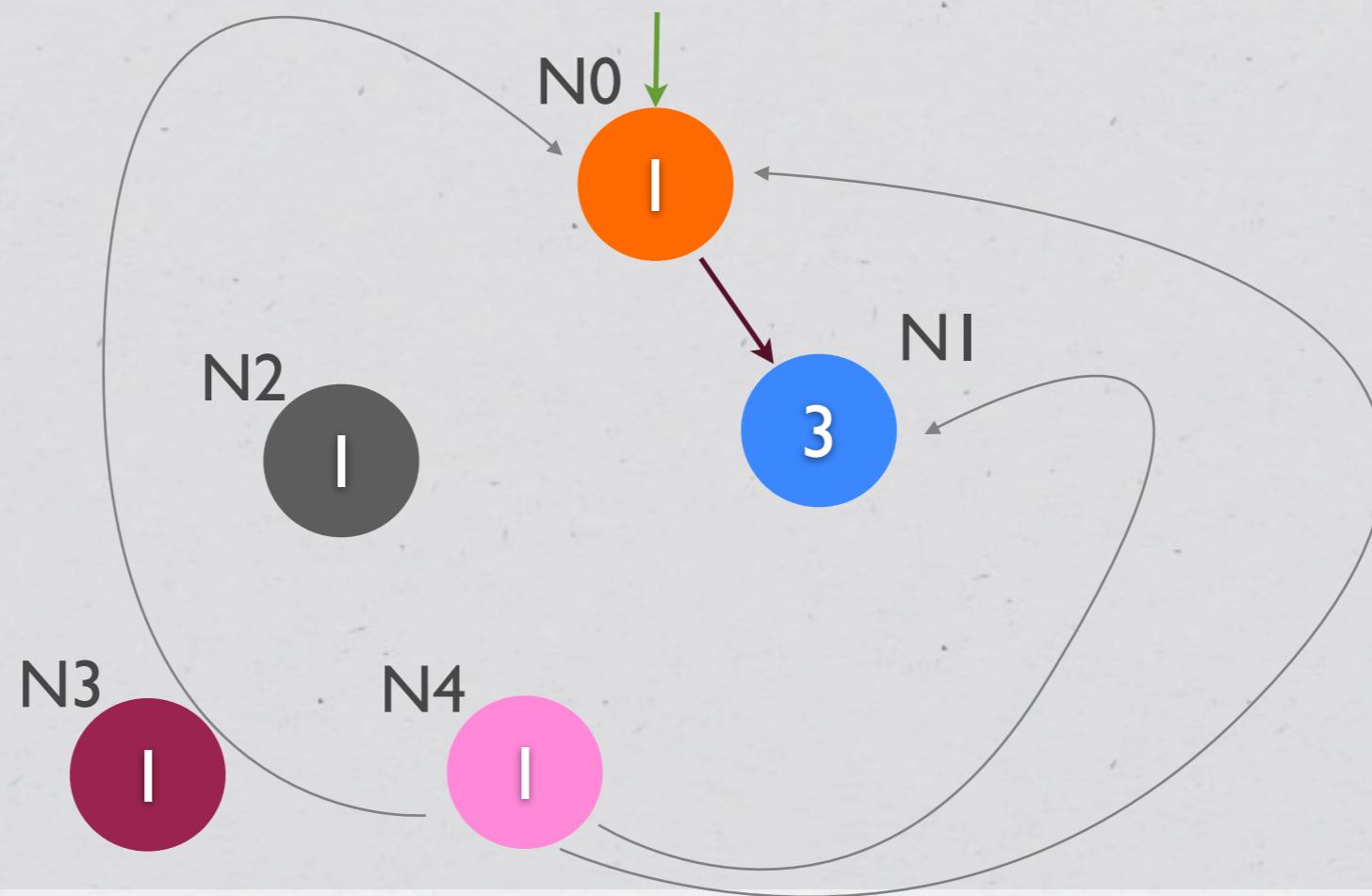
- * Con $n=3$ y $m=3$ obtenemos 3.538.944 estructuras posibles.
- * 16 Árboles rojos y negros válidos.

Problemas en la Explosión de Candidatos Posibles

- * Algunos problemas en el manejo de la explosión de estructuras posibles son los siguientes:
- * Iteración sobre elementos irrelevantes del espacio de estados de la estructura (e.g., sobre elementos inalcanzables del heap)
- * Estructuras simétricas (i.e., instancias redundantes)

Un Ejemplo: Iteración sobre Elementos no Alcanzables

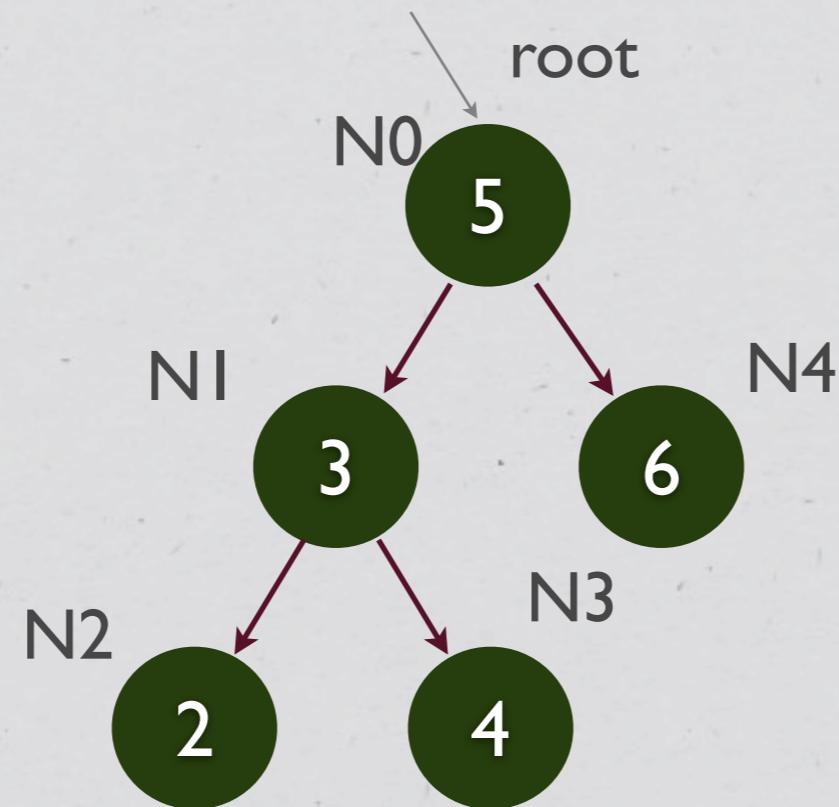
N0	I	#	NI	3	#	#	I	#	#	I	#	#	I	N0	NI
----	---	---	----	---	---	---	---	---	---	---	---	---	---	----	----



Estructuras Simétricas



* Consideremos el siguiente árbol:



* Si nos abstraemos de las direcciones específicas de los nodos, un total de **5!** estructuras diferentes representan el mismo árbol.

Posibles Soluciones

- * En lugar de hacer el backtracking sobre todo el vector candidato, podemos hacer sobre los campos visitados por el repOK cuando se chequea si la estructura es o no válida. De esta manera se evita iterar sobre elementos no alcanzables en el heap.
- * Definir un orden entre los nodos, es decir un orden el forma en la que los nodos aparecen en la estructura. De esta manera evitar generar/visitar estructuras redundantes.

Aún así...

- * En muchos casos el espacio de búsqueda es extremadamente amplio, incluso para cotas de tamaño pequeño.
- * En muchos casos, las test suite producidas, inclusive para cotas pequeñas, son muy grandes. Ejecutarlas resulta prohibitivo.

Algunas Alternativas

- * Usar generación exhaustiva solo con cotas pequeñas y reducir dramáticamente las chances de encontrar bugs!
- * Elegir algunas entradas al azar.
- * Utilizar mecanismo para reducir test suites generadas exhaustivamente y para generarlas más eficientemente.

Algunas Alternativas Estudiadas

- * Utilizar Criterios de cobertura de casos de test para eliminar casos de test que cubren clases ya cubiertas en suites de test previamente generadas.
- * Distinguir porciones de la estructura que son disyuntas en el heap y generarlas por separado (podar el espacio de búsqueda)
- * Podar el espacio búsqueda eliminando candidatos mal formados, utilizando la información provista por el invariante de representación y el mecanismo de rotura de simetría

Utilizar Criterios de Cobertura para achicar suites de test

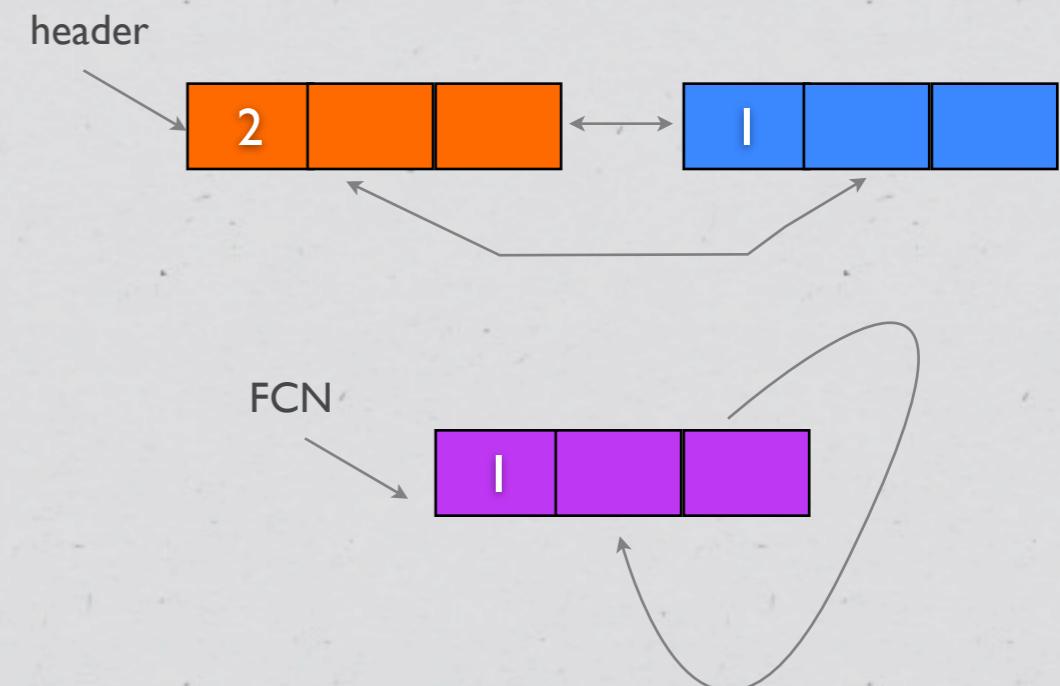
- * Basado en el uso de criterios de cobertura de código sobre el invariante de representación de la estructura para definir clases de equivalencias de las entradas.
- * Esta técnica se basa en la definición de un nuevo criterio de cobertura de Caja Negra en términos de algún criterio de Caja Blanca sobre el código de repOK
- * Podemos considerar que dos casos de test pertenecen a la misma clase de equivalencia si ejercitan (según algún criterio de cobertura de Caja Blanca) de la misma manera el código del repOK() (Invariante de representación)
- * Esta relación de equivalencia entre las entradas válidas es aprovechada para descartar casos de test que cubren clases de equivalencias ya cubiertas por algún otro caso de test

Generación Disyunta

- * Distinguir porciones de la estructura que son disyuntas en el heap.
- * Descomponer el Invariante de representación en invariantes de representación que caracterizan esas subestructuras disyuntas
- * Construir porciones disyuntas del heap de manera separada y luego unirlas.

Generación Disyunta (NodeCachingLinkedList)

```
public class NodeCachingLinkedList {  
    private LinkedListNode header;  
    private int size;  
  
    private LinkedListNode firstCachedNode;  
    private int cacheSize;  
    private int maximumCacheSize;  
    ...  
}  
  
public class LinkedListNode {  
    Object value;  
    LinkedListNode previous;  
    LinkedListNode next;  
    ...  
}
```



header	size	FCN	CS	MCS	value	prev	next	value	prev	next	value	prev	next
N0	2	N2	I	3	2	N0	NI	I	N0	N0	I	#	N2

Otra Alternativa



- * Podar el espacio búsquedas eliminando candidatos mal formados, utilizando la información provista por el invariante de representación y el mecanismo de rotura de simetría
- * En clases con invariantes de representación complejas, muchos asignaciones de valores a campos nos llevan a instancias mal formadas.
- * por ejemplo, sabemos que un árbol binario cuenta con la propiedad de aciclicidad, por lo tanto:

$N0.left \neq N0$

$N0.right \neq N0$

$N1.left \neq N1$

$N1.right \neq N1$

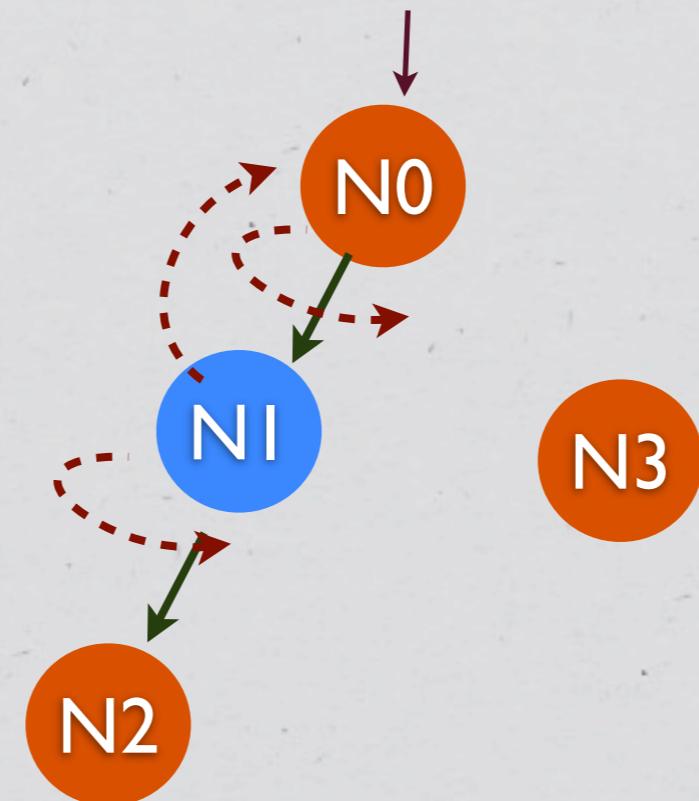
...

Otra Alternativa (Continuación)

- * Los mecanismos de rotura de Simetría también contribuyen a que muchas asignaciones de valores a campos resulten en estructuras inválidas:
 - * por ejemplo, en un árbol binario la raíz no puede ser N1. (o otro nodo distinto de N0 ó null).

Otra Alternativa (Continuación)

*Si contáramos con esta información de antemano podríamos utilizarla para eliminar candidatos mal formados del espacio de búsqueda.



Trabajos Futuros

- * Pensamos continuar trabajando sobre estas líneas de investigación, en particular sobre la última de las técnicas presentadas.
- * Con respecto a Generación Diyunta, evaluar nuevas técnicas para distinguir partes disyuntas del heap y factorización de repOK.

Cotas Ajustadas (Ejemplo)

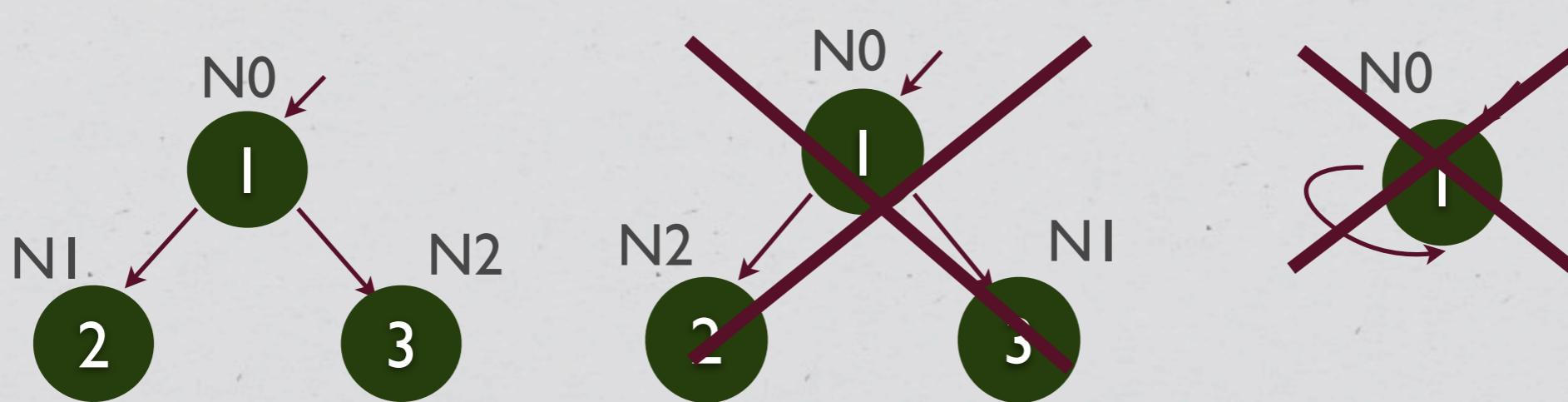


* Consideremos árboles binarios de tamaño máximo 3:

Los campos left y right pueden ser vistos como funciones:

$\text{left}, \text{right} : \{\text{N0}, \text{N1}, \text{N2}\} \rightarrow \{\text{N0}, \text{N1}, \text{N2}, \text{null}\}$, ej: $\text{left}(\text{N0}) \rightarrow \{\text{N0}, \text{N1}, \text{N2}, \text{null}\}$.

podríamos acotarla a: **left(N0) -> {N1,null}**



Incorporación de Cotas Ajustadas en la generación exhaustiva acotada

- * Segundo Algoritmo (basado en el algoritmo de Bounded Lazy Initialization): etiquetar los nodos con un conjunto de nombres de nodos compatibles con las cotas.

root -> {null, N0}

left(N0)-> {null, N1}

left(N1)-> {null, N2, N3}

left(N2)-> {null, N3}

left(N3)-> {null}

right(N0)-> {null, N1, N2}

right(N1)-> {null, N2, N3}

right(N2)-> {null, N3}

right(N3)-> {null}

