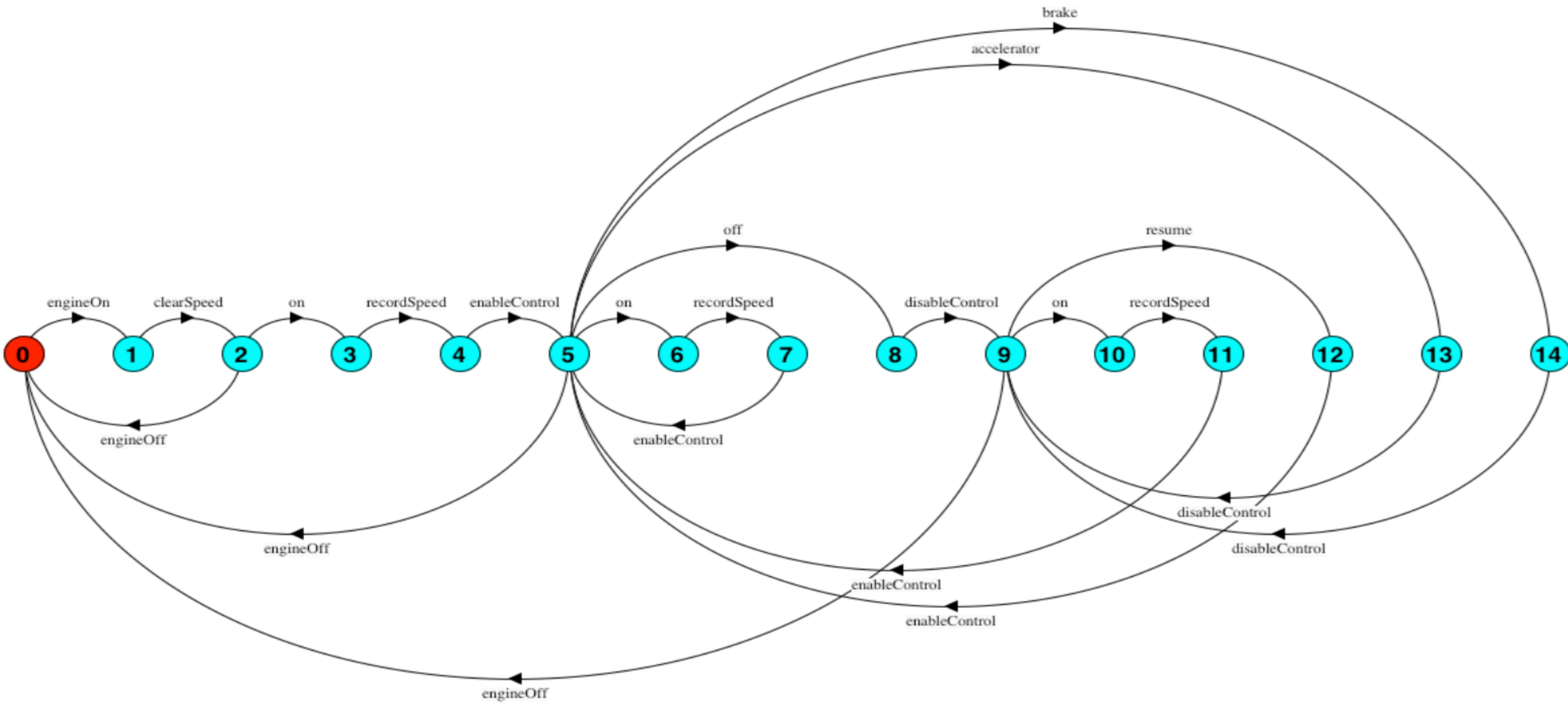
Verificación de Programas Concurrentes (1)

Ingeniería del Software 2

LTS: Labeled Transition System



Definición. *(LTS)* Sea $Estados$ el universo de estados, Act el universo de acciones observables, y $Act_\tau = Act \cup \{\tau\}$. Un LTS es una tupla $P = (S, A, \Delta, s_0)$, donde $S \subseteq Estados$ es un conjunto finito, $A \subseteq Act_\tau$ es un conjunto de etiquetas, $\Delta \subseteq (S \times A \times S)$ es un conjunto de transiciones etiquetadas, y $s_0 \in S$ es el estado inicial. Definimos el alphabeto de comunicacion de P como $\alpha P = A \setminus \{\tau\}$.

Linear Temporal Logic (LTL) - Sintaxis

- p,q,r,... proposiciones atómicas
- $\neg\alpha, \alpha \vee \beta, \alpha \wedge \beta$
- []α (*always*), <>α (*eventually*), Xα (*next*)
- α U β (*until*)
- α R β (*release*)

Linear Temporal Logic (LTL) - Semántica

- σ[i] ⊧ p ssi v(σ[i],p)=**true**
- σ[i] ⊧ Xα ssi σ[i+1] ⊧ α
- σ[i] ⊧ $<>\alpha$ ssi existe j$\geq$i tal que σ[j] ⊧α
- σ[i] ⊧[]α ssi para todo j$\geq$i tal que σ[j] ⊧α
- σ[i] ⊧ αUβ ssi existe k$\geq$i tal que σ[k] ⊧β y para todo j, $i \leq j < k$ se cumple que σ[k] ⊧α
- σ[i] ⊧ α R β ssi o bien σ[i] ⊧ β U ($\alpha \wedge \beta$) o bien σ[i] ⊧[]β

Objetivo

- Dado:
  - Un LTS M representando un programa concurrente
  - Una fórmula LTL P que queremos ver que cumple el programa concurrente
- Definir un algoritmo que retorna **true** si todas las trazas de M satisfacen P

Algoritmo

- Entrada: LTL P, LTS M

1. Convertir la fórmula LTL ¬P a un autómata $A_{\neg P}$ que caracteriza todas las trazas que satisfacen ¬P

2. Chequear que si las trazas de M son disjuntas con las trazas de $A_{\neg P}$

3. Si la intersección es vacía, retornar True, caso contrario devolver traza como contra-ejemplo.

1-Convertir LTL a Autómata

(¿Pero qué tipo autómata?)

- Una fórmula LTL caracteriza un conjunto de trazas (las trazas que satisfacen la fórmula)
- **Objetivo**: Construir un autómata A_P cuyo lenguaje es el mismo que una formula LTL P.
- **Pregunta**: ¿Qué tipo de autómata conocemos de <u>teoría de lenguajes</u> que pueda reconocer trazas *infinitas?*

Autómatas para Trazas Infinitas

Los AFD (ver definición de TLA) aceptan lenguajes regulares

Definición 1 (Autómata Finito Determinístico (AFD)). *Es una 5-upla* $\langle Q, \Sigma, \delta, q_0, F\rangle$ *donde*

- Q *es un conjunto finito de estados*
- Σ *es un conjunto finito de símbolos que constituye el alfabeto de entrada*
- $\delta : Q \times \Sigma \to Q$ *es la función de transición*
- $q_0 \in Q$ *es el estado inicial*
- $F \subseteq Q$ *es el conjunto de estados finales*

Los lenguajes regulares tienen trazas finitas

Definición 4 (Lenguaje aceptado por un AFD). *Dado un AFD* $M = \langle Q, \Sigma, \delta, q_0, F\rangle$*, el lenguaje aceptado por* M*,* $\mathcal{L}(M)$*, es el conjunto de cadenas aceptadas por* M *y se define como*

$$\mathcal{L}(M) = \left\{ x \in \Sigma^* : \widehat{\delta}(q_0, x) \in F \right\}.$$

Necesitamos otro tipo de reconocedor de lenguajes

LTL Properties ≡ Büchi automata

[Vardi and Wolper LICS 86]

- **Büchi automata**: Autómata de estados finitos que reconocen lenguajes de palabras infinitas, más específicamente lenguajes ω-regulares

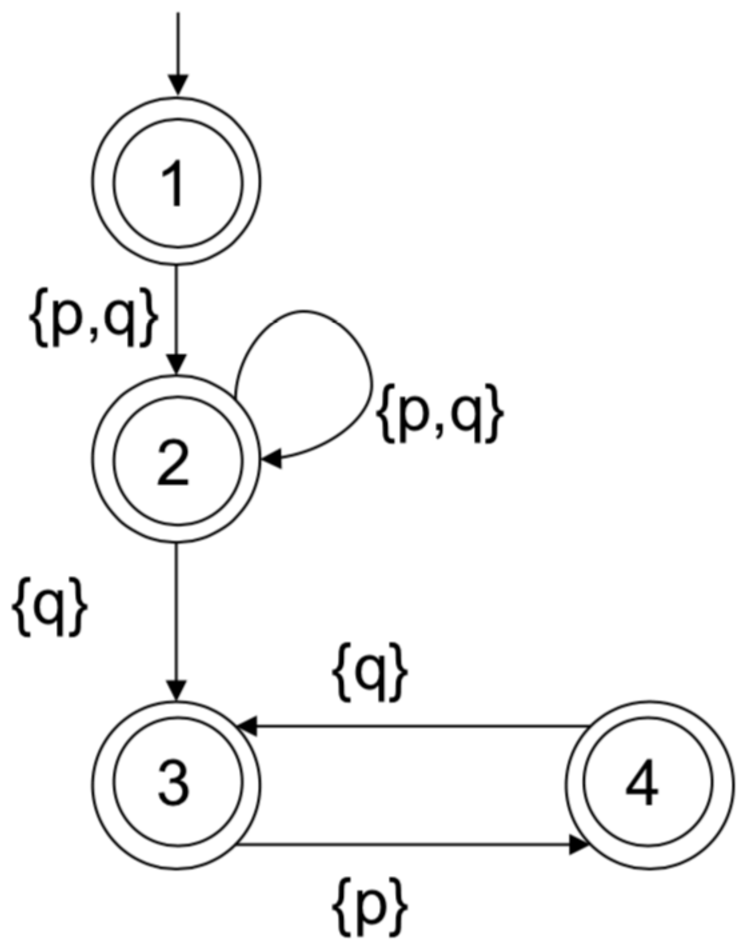
- **Büchi automata** acepta una cadena cuando su ejecución en el autómata visita un estado de aceptación infinitas veces.

- Formulas LTL pueden ser traducidas a Büchi. El autómata acepta una una traza si y solo si esa traza satisface la fórmula

Autómata de Büchi

- Un autómata de Büchi es una tupla A = $\langle \Sigma, Q, \Delta, Q_0, F \rangle$ donde
  - Σ es un alfabeto finito
  - Q es un conjunto finito de estados
  - $\Delta \subseteq Q \times \Sigma \times Q$ es la relación de transición
  - $Q_0 \subseteq Q$ es el conjunto de estados iniciales
  - $F \subseteq Q$ es el conjunto de estados de aceptación

Ejemplo: Autómata de Büchi



- ¿Cuánto valen $\Sigma, Q, \Delta, Q_0, F$ en este ejemplo?

Lenguaje de un Autómata de Büchi

- Un autómata de Büchi reconoce un lenguaje que consiste en un conjunto de secuencias infinitas sobre el alfabeto Σ

- Sea A un autómata de Büchi, sea el lenguaje aceptado por el autómata de Büchi $L(A)$, y sea Σ^ω el conjunto de todas las secuencias infinitas sobre Σ.

  - Entonces vale que $L(A) \subseteq \Sigma^\omega$

Lenguaje de un Autómata de Büchi

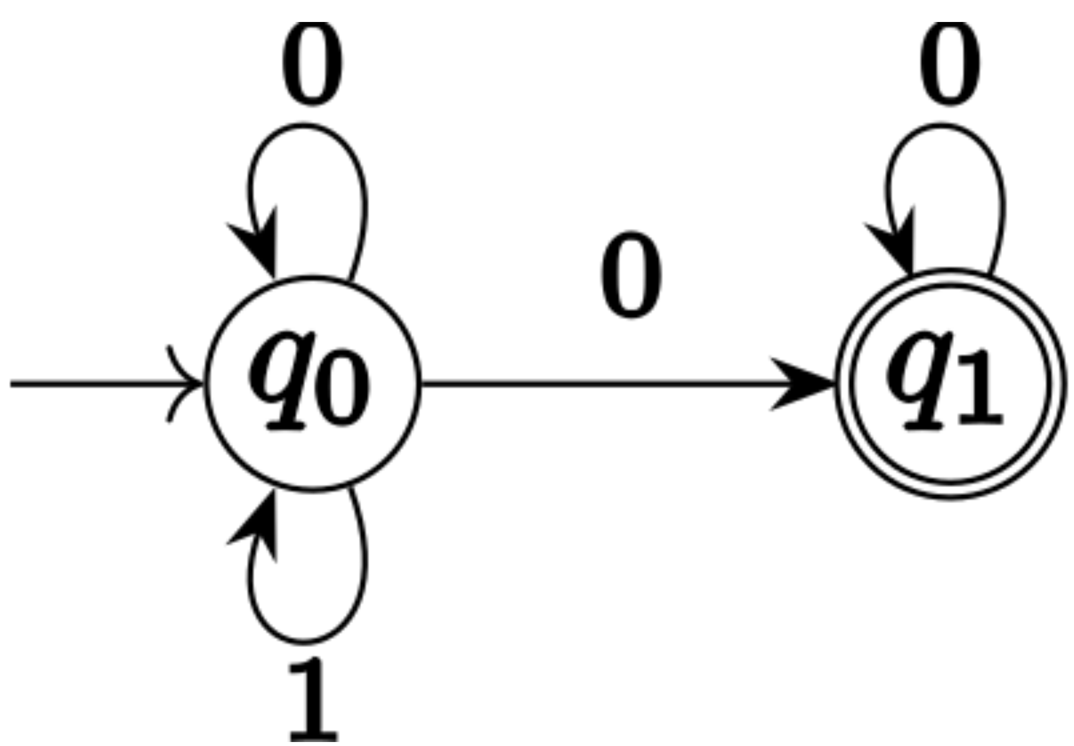
- Dada una secuencia infinita $w \in \Sigma^\omega$ donde $w = a_0 a_1 a_2 \ldots$ una ejecución r del automáta A sobre w es una secuencia de estados $r = q_0 q_1 q_2 \ldots$ donde $q \in Q_0$ y para cada $i \geq 0$ vale que $\langle q_i, a_i, q_{i+1} \rangle \in \Delta$

- Dada una ejecución r, sea $inf(r) \subseteq Q$ el conjunto de estados del autómata que aparecen en r una cantidad infinita de veces

- Una ejecución r es una "ejecución aceptada" si y solamente si $inf(r) \cap F \neq \emptyset$

  - En otras palabras, r es una "ejecución aceptada" si atraviesa al menos un estado de aceptación una cantidad finita de veces

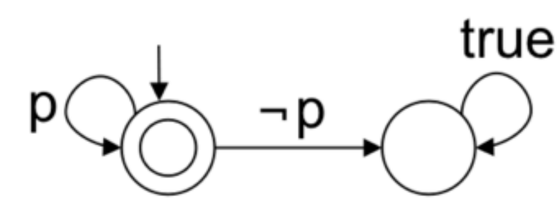
Ejemplo: Autómata de Büchi



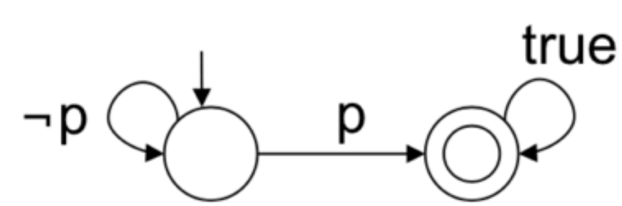
- ¿Qué cadenas infinitas acepta este autómata?
  - ¿Acepta 0...? Sí
  - ¿Acepta 0101..? No
- En general, acepta el lenguaje $(0 \mid 1)^*0^\omega$

Ejemplos LTL2Buchi

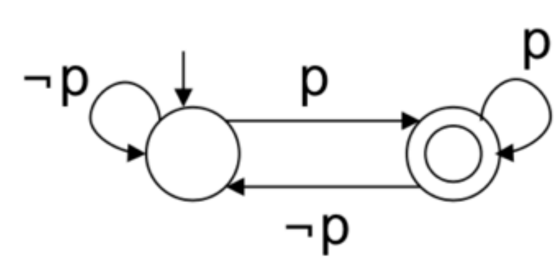
□p



<>p



□(<>p)



El tamaño del autómata crece exponencialmente con respecto al tamaño de la fórmula.

Autómata de Büchi Generalizado

- Un autómata de Büchi **generalizado** es una tupla A = $\langle \Sigma, Q, \Delta, Q_0, F \rangle$ donde
  - Σ es un alfabeto finito
  - Q es un conjunto finito de estados
  - $\Delta \subseteq Q \times \Sigma \times Q$ es la relación de transición
  - $Q_0 \subseteq Q$ es el conjunto de estados iniciales
  - $F \subseteq 2^Q$ es el conjunto de conjuntos de estados de aceptación

Lenguaje de un Automáta de Büchi Generalizado

- Dada una secuencia infinita $w \in \Sigma^\omega$ donde $w = a_0 a_1 a_2 \ldots$ una ejecución r del automáta A sobre w es una secuencia de estados $r = q_0 q_1 q_2 \ldots$ donde $q \in Q_0$ y para cada $i \geq 0$ vale que $\langle q_i, a_i, q_{i+1} \rangle \in \Delta$

- Dada una ejecución r, sea $inf(r) \subseteq Q$ el conjunto de estados del autómata que aparecen en r una cantidad infinita de veces

- Una ejecución r es una "ejecución aceptada" si y solamente si $inf(r) \cap F_i \neq \emptyset$ para todo $F_i \in F$

    - En otras palabras, r es una "*ejecución aceptada*" si atraviesa al menos un estado de aceptación de cada **conjunto de aceptación** una cantidad finita de veces

Automáta de Büchi vs. Autómata de Büchi Generalizado

- Un autómata de Büchi A = $\langle \Sigma, Q, \Delta, Q_0, F \rangle$ se puede transformar trivialmente en un autómata de Büchi generalizado A'=$\langle \Sigma, Q, \Delta, Q_0, \{F\} \rangle$

- Ahora, transformar un autómata de Büchi generalizado en un autómata de Büchi no-generalizado (o común) no es tan directo (ya lo vamos a ver)

De LTL a Büchi

Un método de construcción basado en Tableau

Tableau para Lógica Proposicional

- Procedimiento para decidir si una formula proposicional es satisfacible

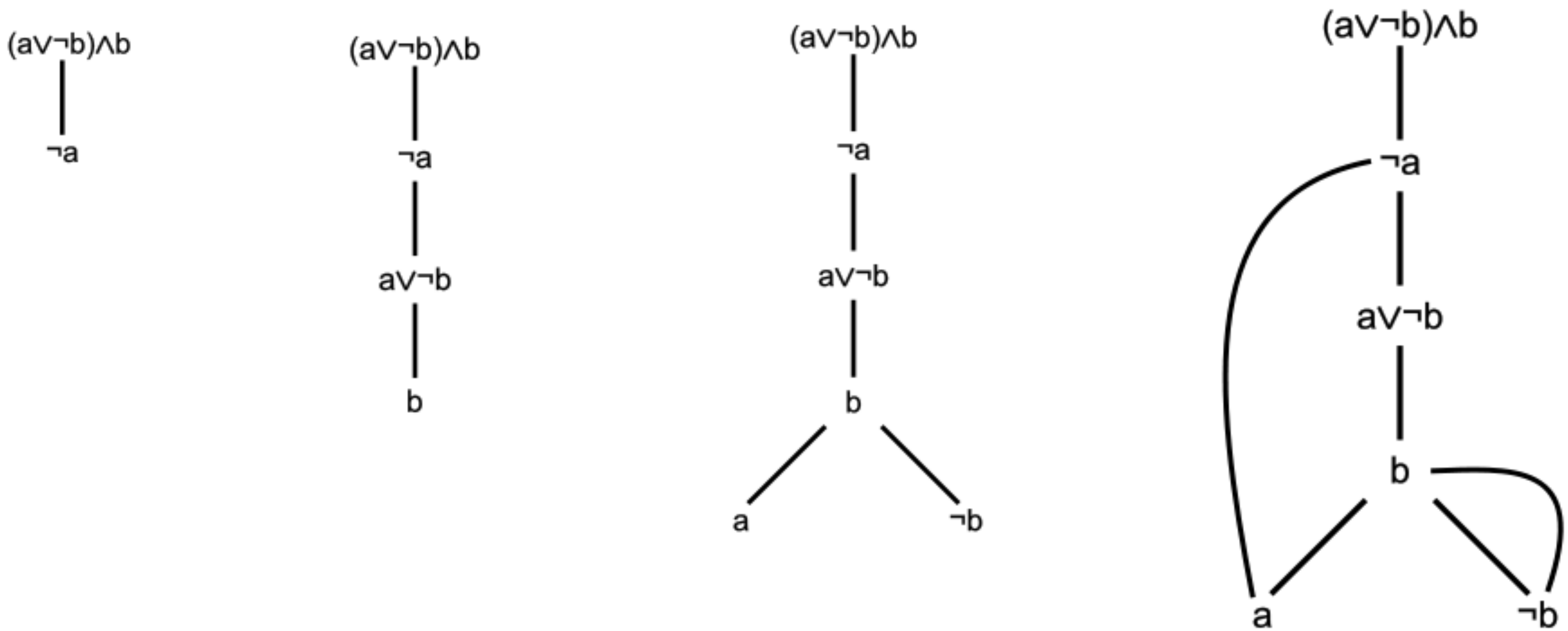
- Idea básica: Descomponer top/down la formula en sub-fórmulas armando un árbol

- Una fórmula es satisfacible si una rama del árbol *no se cierra.*

Reglas de Descomposición

$A \wedge B$ A B	$A \vee B$ A B	$A \rightarrow B$ $\neg A$ B
$A \leftrightarrow B$ $A \wedge B$ $\neg A \wedge \neg B$	$\neg\neg A$ A	$\neg(A \wedge B)$ $\neg A$ $\neg B$
$\neg(A \vee B)$ $\neg A$ $\neg B$	$\neg(A \rightarrow B)$ A $\neg B$	$\neg(A \leftrightarrow B)$ $A \wedge \neg B$ $\neg A \wedge B$

Ejemplo: $\neg a \wedge ((a \vee \neg b) \wedge b)$



LTL Tableau

- Objetivo: Construir un autómata de Büchi que acepta el mismo lenguaje que una fórmula LTL dada.

- Idea:

    - Que cada estado del autómata de Büchi sepa qué formula debe reconocer

    - Que al avanzar de s a s' por a, la fórmula de s' sea como la de s al después de haber "procesado" a.

Esto sería "similar" a la idea de descomposición de tableau proposicional

LTL2Büchi - El Algoritmo

[Gerth, Peled, Vardi, Wolper 95]

- **Input**: Una formula LTL en forma normal positiva *(sólo proposiciones pueden estar negadas)*
- **Output**: Un Büchi que reconoce el mismo lenguaje

- $\neg(\alpha \cup \beta) \equiv \neg\alpha R \neg\beta$
- $\neg(\alpha R \beta) \equiv \neg\alpha \cup \neg\beta$
- $\neg(X\alpha) \equiv X\neg\alpha$
- $\neg(\alpha R \beta) \equiv \neg\alpha \cup \neg\beta$
- $[]\alpha \equiv true \cup \alpha$
- $<>\alpha \equiv \text{false}\ R\alpha$

LTL2Büchi - El Algoritmo

[Gerth, Peled, Vardi, Wolper 95]

- Cada estado tendrá tres conjuntos de propiedades
  - ***New***: Las propiedades que deben valer desde el estado pero que no fueron “procesadas” por el algoritmo
  - ***Old***: Las propiedades que deben valer desde el estado y que ya fueron “expandidas” por el algoritmo
  - ***Next***: Las propiedades que deben valer en los estados sucesores inmediatos.
- Además, cada estado tendrá una lista de estados
  - ***Incoming***: Los estados predecesores inmediatos

LTL2Büchi - El Algoritmo

[Gerth, Peled, Vardi, Wolper 95]

Input: Formula LTL P

(Una intuición)

1. Crear un nodo n=<New = {P}, Old =∅, Next=∅, Incoming= ∅>

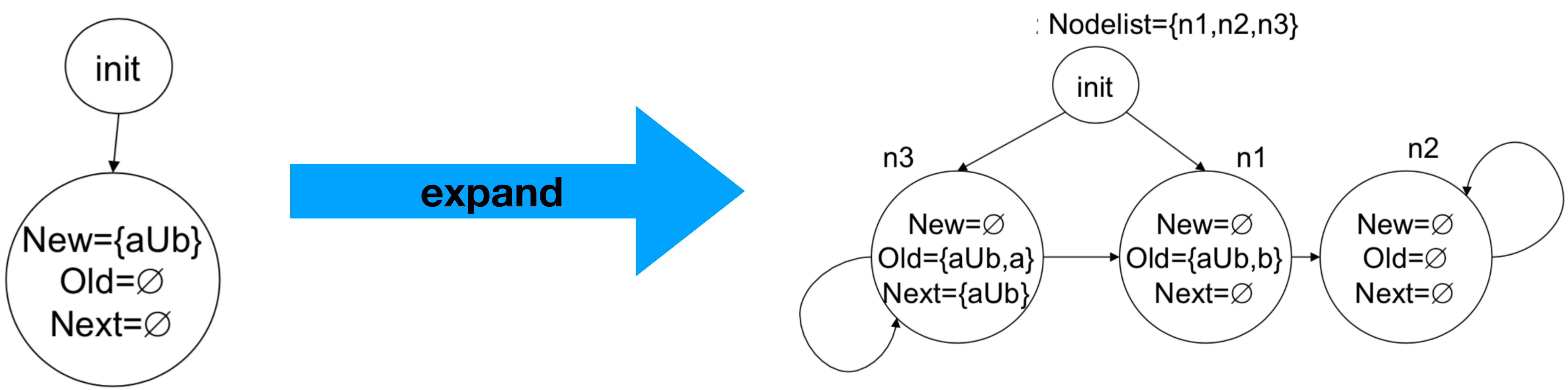
2. Para cada nodo n con f ∈ new, procesar f creando nuevos nodos. Continuar hasta que no exista f ∈ new en ningún nodo n.

3. Construir un autómata de Büchi generalizado a partir del autómata

4. Traducir el Büchi generalizado en un Büchi común

Ejemplo: a U b

NodeList=∅



LTL2Büchi - El Algoritmo

Inicialmente, el conjunto de nodos "ya procesados" empieza vacío

```
TranslateLTL2Buchi(f) {
  expand(<Incoming:={init}, Old:=∅, New:={f}, Next:=∅>, ∅)
}
```

Arrancamos el algoritmo de traducción invocando a la rutina auxiliar "expand" con un nodo inicial cuyo New es la fórmula f y una lista vacía de nodos procesados

Caso Reuso

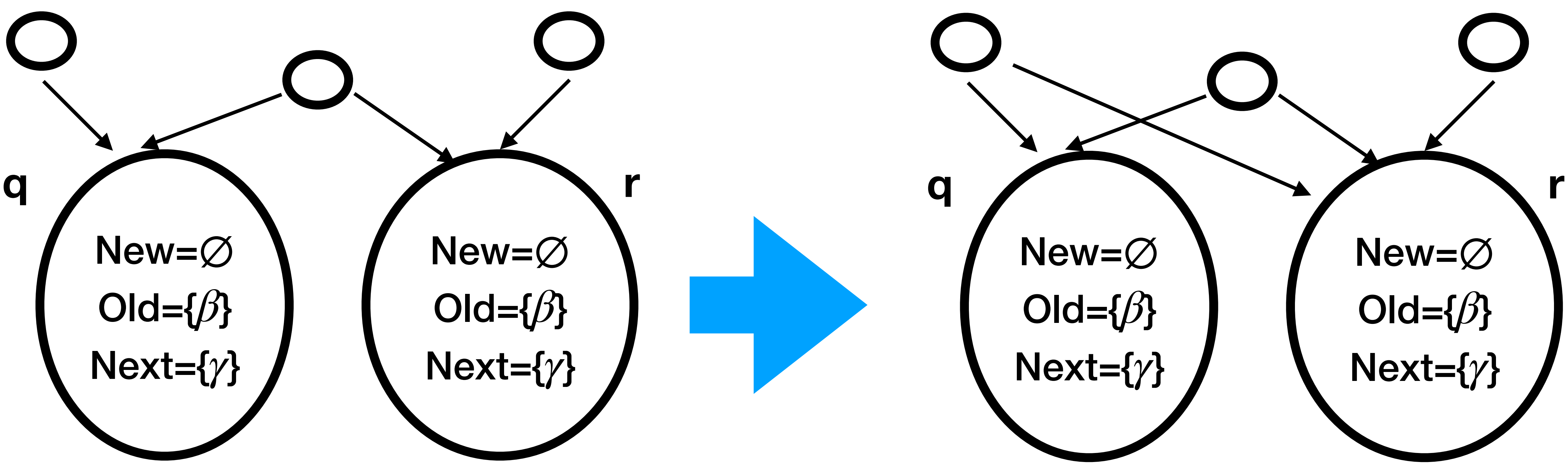
La función auxiliar "expand" recibe un nodo q a ser procesado y un conjunto "NodeList" de nodos ya procesados

Si New(q) es vacío significa que no quedan fórmulas en q para procesar

```
expand(q, NodeList) {
  If New(q)=∅
  Then
     If r ∈ NodeList s.t. Old(r)=Old(q) and Next(r)=Next(q)
     Then
        Incoming(r) := Incoming(q) ∪ Incoming(r)
        return (NodeList)
     Else
        ...
```

Si el nodo q no posee fórmulas a ser procesadas en "New", y ademas ya existe otro nodo procesado con el mismo Old y Next, entonces "colapsamos" q en r (i.e. reusamos el nodo), ya que representan el mismo comportamiento

```
expand(q, NodeList) {
  If New(q)=∅
  Then
     If r ∈ NodeList s.t. Old(r)=Old(q) and Next(r)=Next(q)
     Then
        Incoming(r) := Incoming(q) ∪ Incoming(r)
        return (NodeList)
     Else
        ...
```



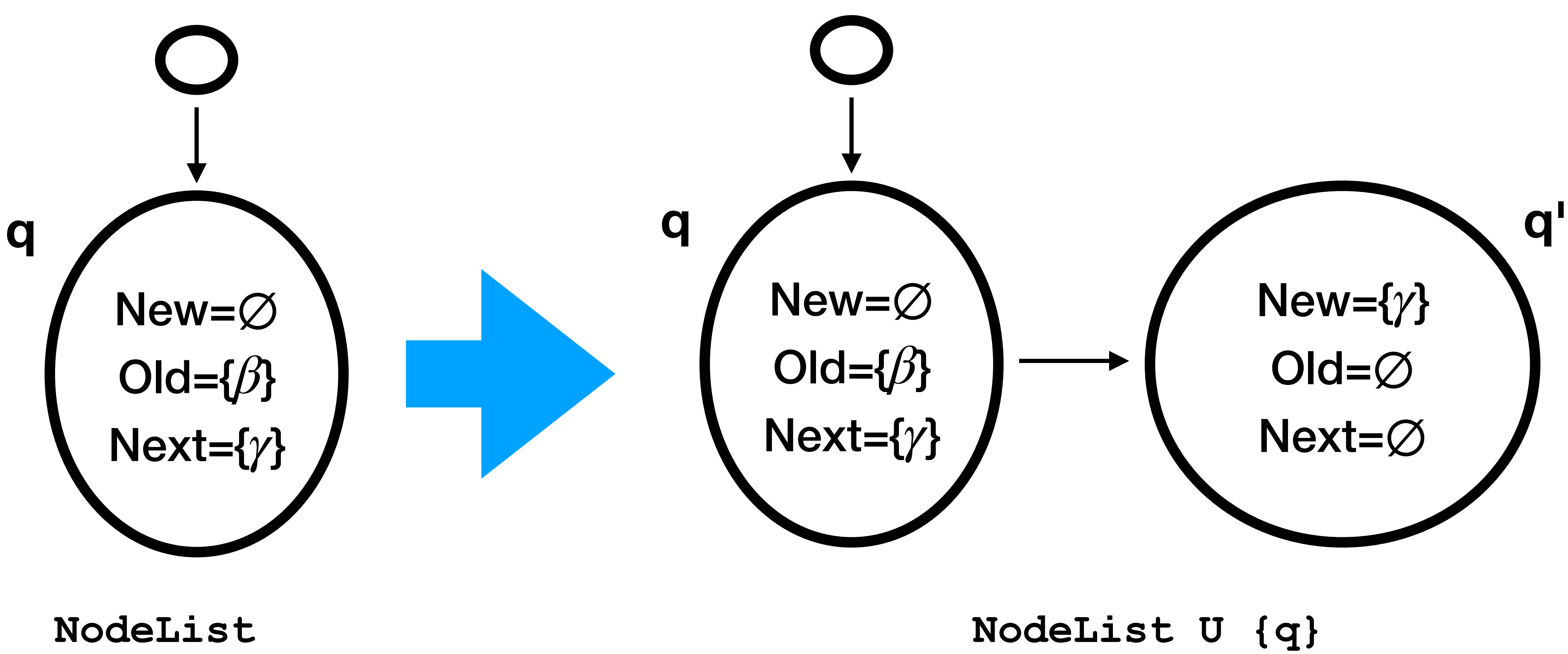
New es vacío y sin reuso

Veamos ahora que pasa si no existe un nodo "r" que podamos reutilizar que ya haya sido procesado

```
expand(q, NodeList) {
  If New(q)=∅
  Then
     If r ∈ NodeList s.t. Old(r)=Old(q) and Next(r)=Next(q)
     Then
        Incoming(r) := Incoming(q) ∪ Incoming(r)
        return (NodeList)
     Else
        Create a new nodo q' s.t.
          Incoming(q') := q
          Old(q') := ∅
          New(q') := Next(q)
          Next(q') := ∅
        Return expand(q', NodeList ∪ {q})
```

En ese caso, creamos un nodo nodo q' tal que "provenga" de q (el nodo que estamos procesando), marcamos para procesar las fórmulas que deben cumplirse en el siguiente estado de q (Next(q)) y marcamos como procesado a "q", llamado a expand recursivamente

```
expand(q, NodeList) {
  If New(q)=∅
  Then
     If r ∈ NodeList s.t. Old(r)=Old(q) and Next(r)=Next(q)
     Then
        Incoming(r) := Incoming(q) ∪ Incoming(r)
        return (NodeList)
     Else
        Create a new nodo q' s.t.
          Incoming(q') := q
          Old(q') := ∅
          New(q') := Next(q)
          Next(q') := ∅
        Return expand(q', NodeList ∪ {q})
```



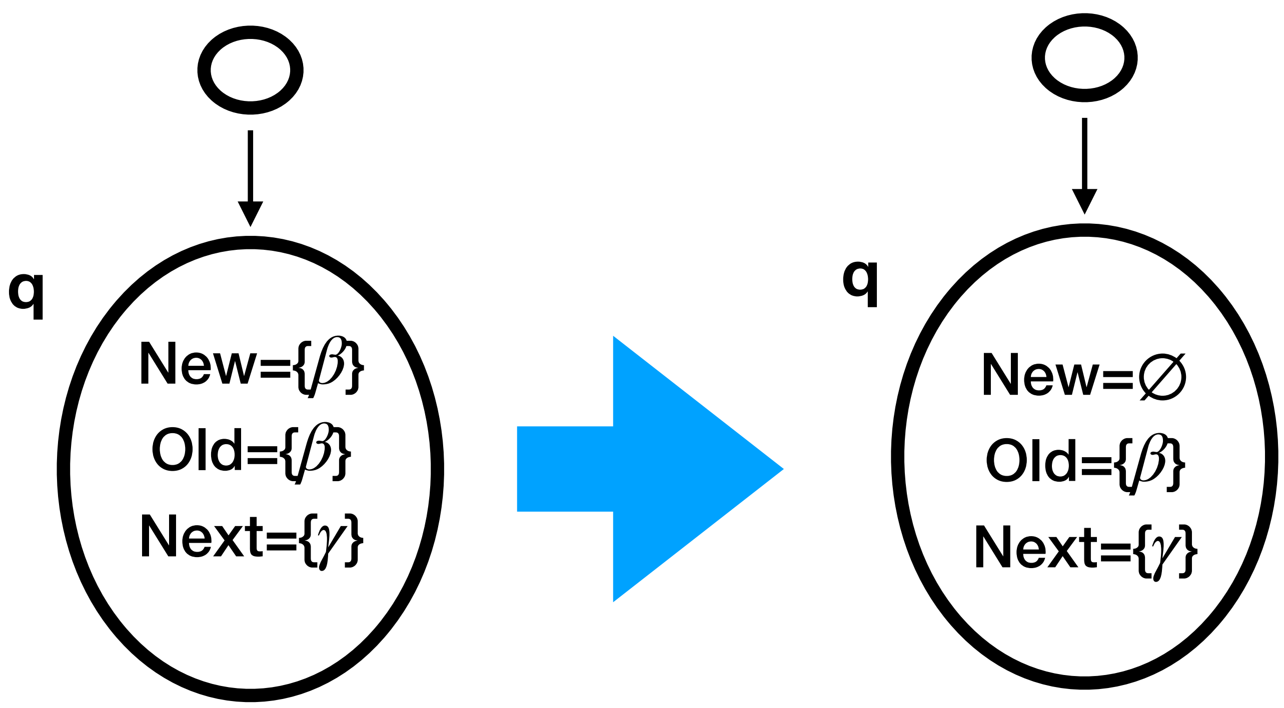
Ahora veamos que pasa si efectivamente existen fórmulas en q que debamos procesar

```
expand(q, NodeList) {
  If New(q)=∅
  Then
    ... // procesamos si New(q) es vacío
  Else
   // New(q) no es vacío
   Pick f ∈ New(q)
   New(q) := New(q) - {f}
   If f ∈ Old(q)
   Then
     Return expand(q,NodeList)
   Else
     ...
```

Seleccionamos alguna fórmula que esté en New(q) y la eliminamos del conjunto New(q)

Si la fórmula ya esta en Old(q) (i.e. ya fue procesada), continuamos la ejecución recursivamente

```
expand(q, NodeList) {
  If New(q)=∅
  Then
    ... // procesamos si New(q) es vacío
  Else
   // New(q) no es vacío
   Pick f ∈ New(q)
   New(q) := New(q) - {f}
   If f ∈ Old(q)
   Then
     Return expand(q,NodeList)
   Else
     ...
```



```
expand(q, NodeList) {
  If New(q)=∅
  Then
    ... // procesamos si New(q) es vacío
  Else
   // New(q) no es vacío
   Pick f ∈ New(q)
   New(q) := New(q) - {f}
   If f ∈ Old(q)
   Then
     Return expand(q,NodeList)
   Else
     ...
```

Ahora, veamos que pasa si f no ha sido todavía procesada en el nodo q por el algoritmo

Para eso, separamos en casos de acuerdo a qué forma tiene la fórmula "f"

f es primitiva/literlal

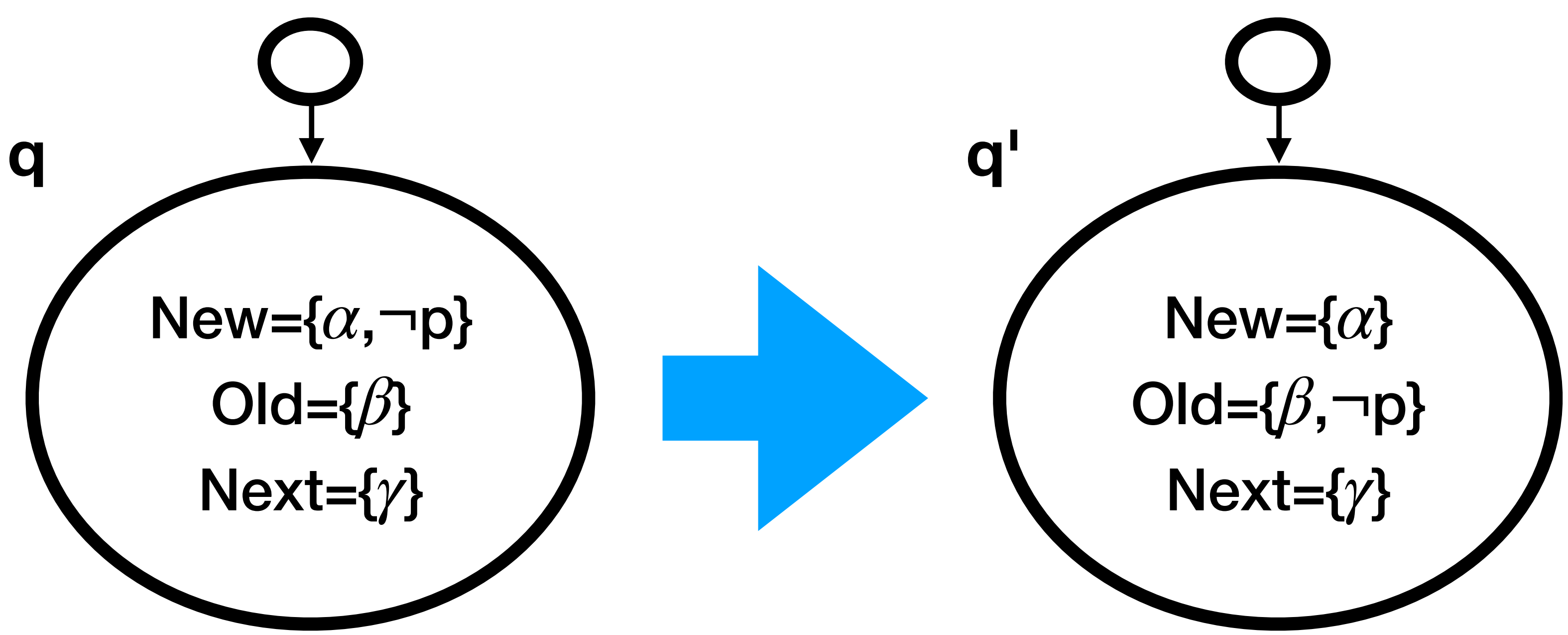
```
If f is a boolean constant or f ∈ AP or ¬f ∈ AP
Then
  If f = false or ¬f ∈ Old(q)
    Then
      Return NodeList
    Else
      Create New Node q' s.t.
        Incoming(q') := Incoming(q)
        Old(q') := Old(q) ∪ {f}
        New(q') := New(q) - {f}
        Next(q') := Next(q)
      Return expand(q', NodeList)
```

Si f es la constante "false" o la negación de f ya fue procesada en q, no tenemos que hacer nada y retornamos la NodeList

En ese caso, reemplazamos el nodo q con un nodo q' donde "pasamos" la fórmula f de "New" a "Old",es decir, marcamos a la fórmula f como "procesada".

f es primitiva/literal

```
If f is a boolean constant or f ∈ AP or ¬f ∈ AP
Then
  If f = false or ¬f ∈ Old(q)
    Then
      Return NodeList
    Else
      Create New Node q' s.t.
        Incoming(q') := Incoming(q)
        Old(q') := Old(q) ∪ {f}
        New(q') := New(q) - {f}
        Next(q') := Next(q)
      Return expand(q', NodeList)
```



$f \equiv h \vee k$

Si f es una disjunción

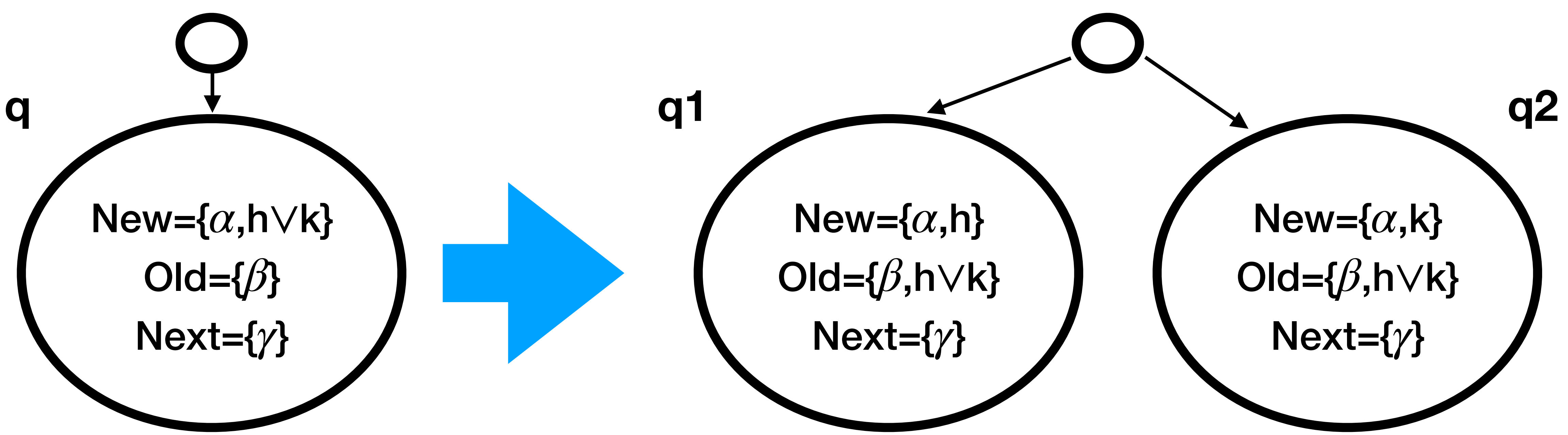
```
ElseIf f≡h∨k
Then
  Create two nodes q1, q2 sucht that
    Incoming(q1):=Incoming(q2):=Incoming(q)
    Old(q1):=Old(q2):=Old(q)∪{h∨k}
    New(q1):=(New(q)-{h∨k})∪{h}
    New(q1):=(New(q)-{h∨k})∪{k}
    Next(q1):=Next(q2):=Next(q)
  Return expand(q2, expand(q1, NodeList)
```

Creo dos nodos q1,q2 tales que comparten el mismo incoming y su Old es Old(q) y la fórmula {h∨h}, donde q1 procesará "h" y q2 procesará "k"

Retorno expandir primero q1, y luego q2.

$f \equiv h \vee k$

```
ElseIf f≡h∨k
Then
  Create two nodes q1, q2 sucht that
    Incoming(q1):=Incoming(q2):=Incoming(q)
    Old(q1):=Old(q2):=Old(q)∪{h∨k}
    New(q1):=(New(q)-{h∨k})∪{h}
    New(q1):=(New(q)-{h∨k})∪{k}
    Next(q1):=Next(q2):=Next(q)
  Return expand(q2, expand(q1, NodeList)
```



$f \equiv h \wedge k$

Si f es una conjunción de h y k

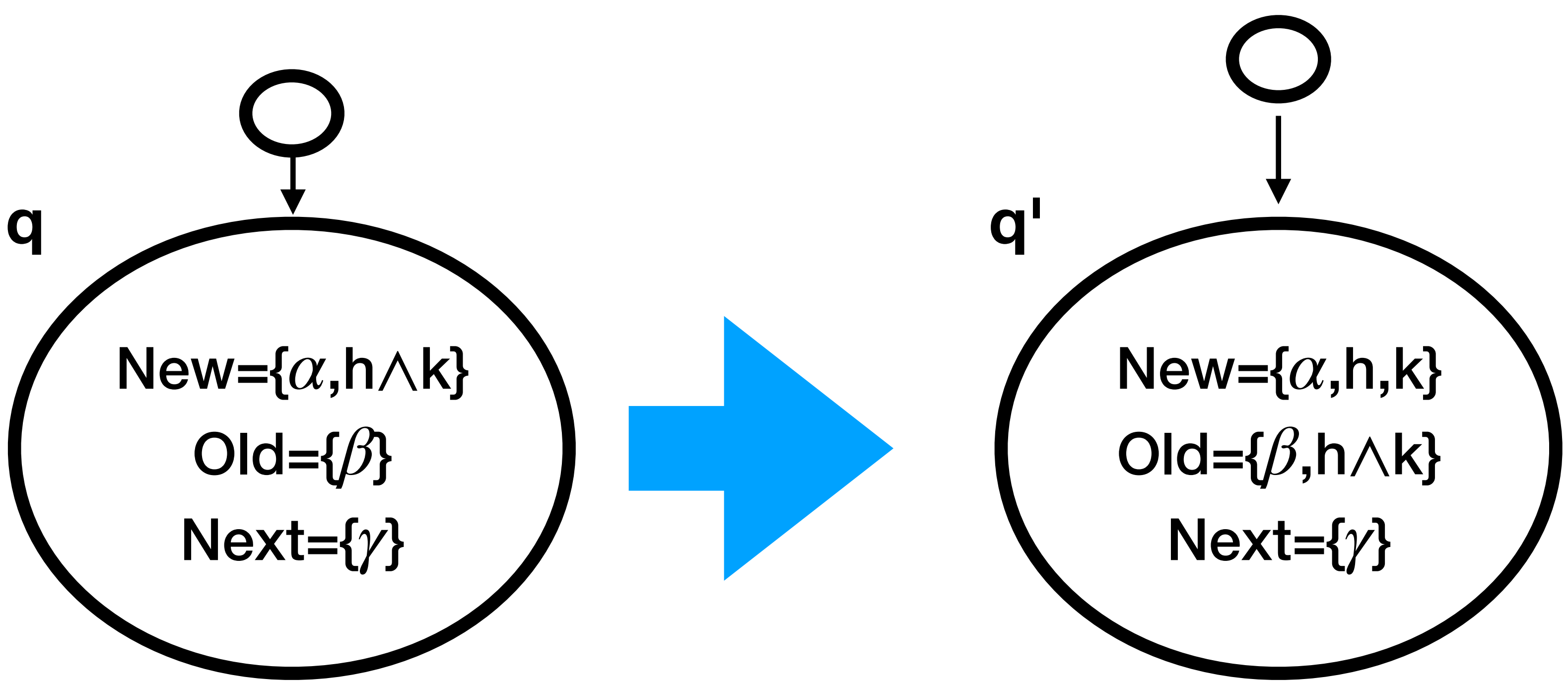
```
ElseIf f≡h∧k
Then
  Create one node q' sucht that
    Incoming(q'):=Incoming(q)
    Old(q'):=Old(q)∪{h∧k}
    New(q'):=(New(q)-{h∧k})∪{h}∪{k}
   Next(q'):=Next(q)
  Return expand(q', NodeList)
```

Creo un nodo q' donde h∧k ya está "procesada", y agregamos para procesar "h" y "k" simultáneamente

Retorno expandir primero q'

f≡h∧k

```
ElseIf f≡h∧k
Then
  Create one node q' sucht that
    Incoming(q'):=Incoming(q)
    Old(q'):=Old(q)∪{h∧k}
    New(q'):=(New(q)-{h∧k})∪{h}∪{k}
   Next(q'):=Next(q)
  Return expand(q', NodeList)
```



f≡X h

Si f es "vale h en el siguiente estado"

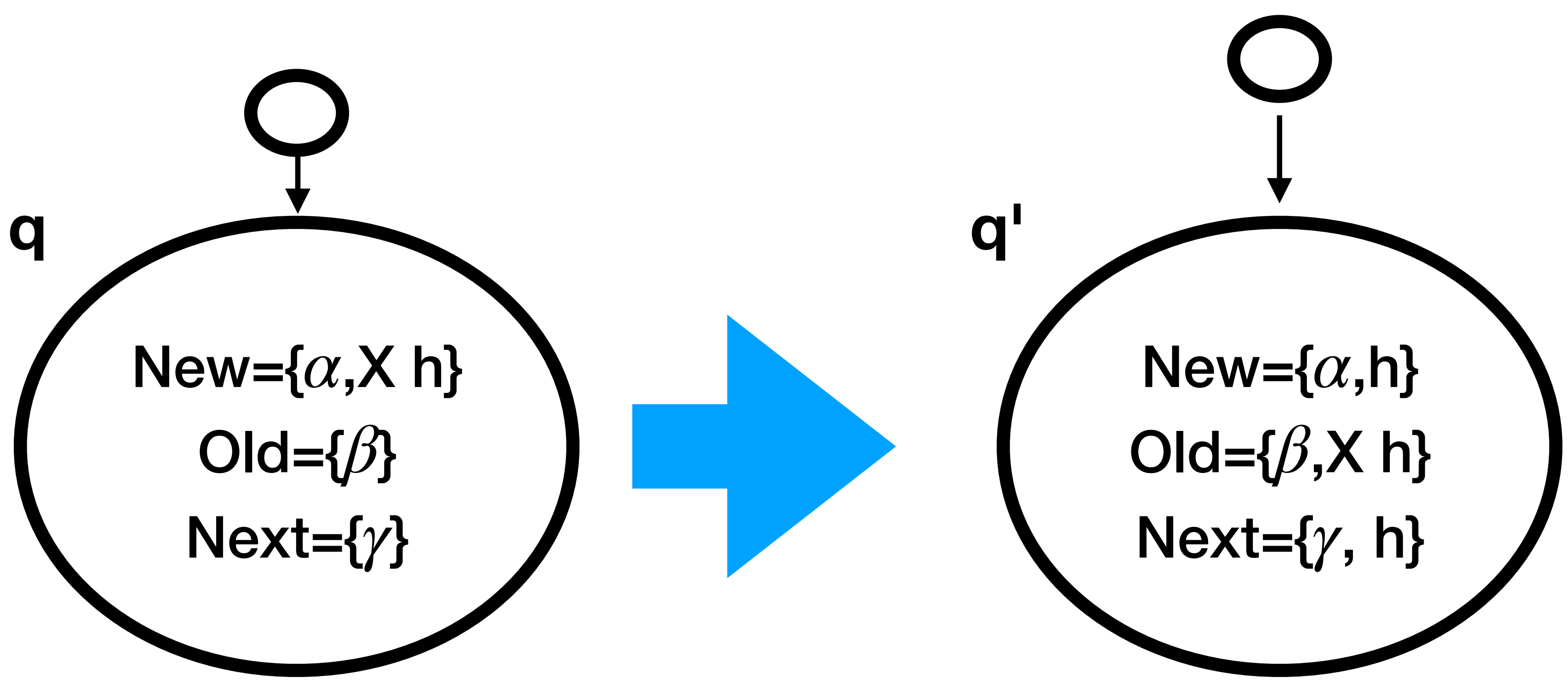
```
ElseIf f≡ X h
Then
  Create one node q' sucht that
    Incoming(q'):=Incoming(q)
    Old(q'):=Old(q)∪{X h}
    New(q'):=New(q)-{X h}
   Next(q'):=Next(q)∪{h}
  Return expand(q', NodeList)
```

Creo un nodo q' donde "X h" ya está "procesada", y agregamos para procesar "h" únicamente

Retorno expandir primero q'

f≡X h

```
ElseIf f≡ X h
Then
  Create one node q' sucht that
    Incoming(q'):=Incoming(q)
    Old(q'):=Old(q)∪{X h}
    New(q'):=New(q)-{X h}
   Next(q'):=Next(q)∪{h}
  Return expand(q', NodeList)
```



$f \equiv h \cup k$

Si f es "vale h hasta que vale k"

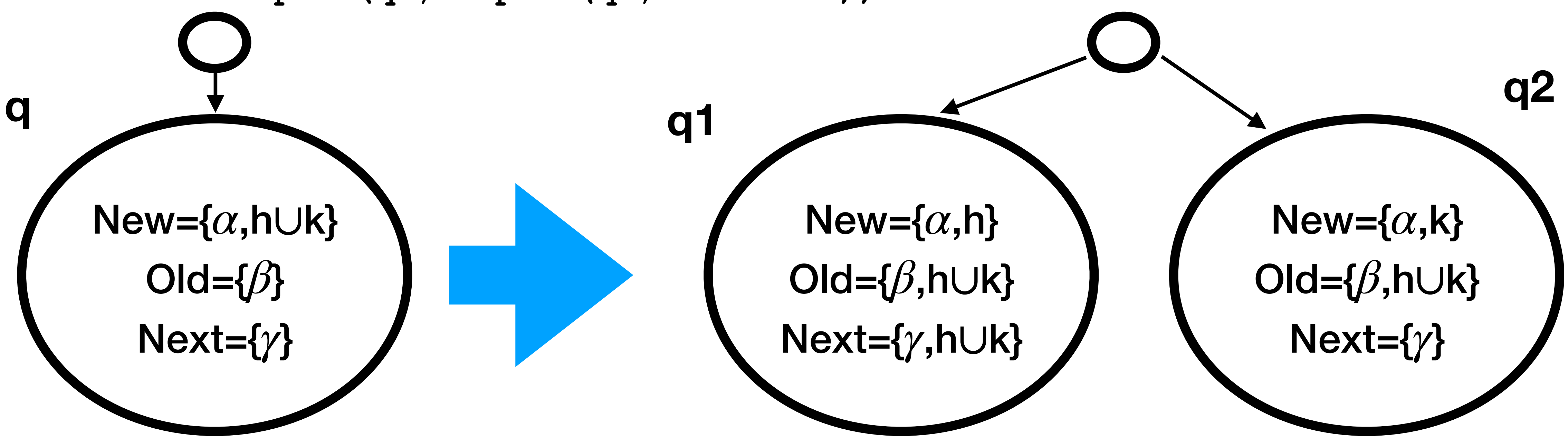
```
ElseIf f≡ h ∪ k
Then
  Create two nodes q1, q2 sucht that
    Incoming(q1):=Incoming(q2):=Incoming(q)
    Old(q1):=Old(q2):=Old(q)∪{h ∪ k}
    New(q1):=New(q)∪{h}
    New(q2):=New(q)∪{k}
    Next(q1):=Next(q)∪{h ∪ k}
    Next(q2):=Next(q)

  Return expand(q2, expand(q1, NodeList))
```

Creo dos nodos distintos q1, q2 donde q1 toma "h" y q2 toma "k", pero si vale "h", entonces en el siguiente nodo debe vale hUk

Retorno expandir primero q1 y luego q2

f≡ h ∪ k

```
ElseIf f≡ h ∪ k
Then
  Create two nodes q1, q2 sucht that
    Incoming(q1):=Incoming(q2):=Incoming(q)
    Old(q1):=Old(q2):=Old(q)∪{h ∪ k}
    New(q1):=New(q)∪{h}
    New(q2):=New(q)∪{k}
    Next(q1):=Next(q)∪{h ∪ k}
    Next(q2):=Next(q)

  Return expand(q2, expand(q1, NodeList))
```



f≡ h R k

Si f es "vale h hasta que vale k"

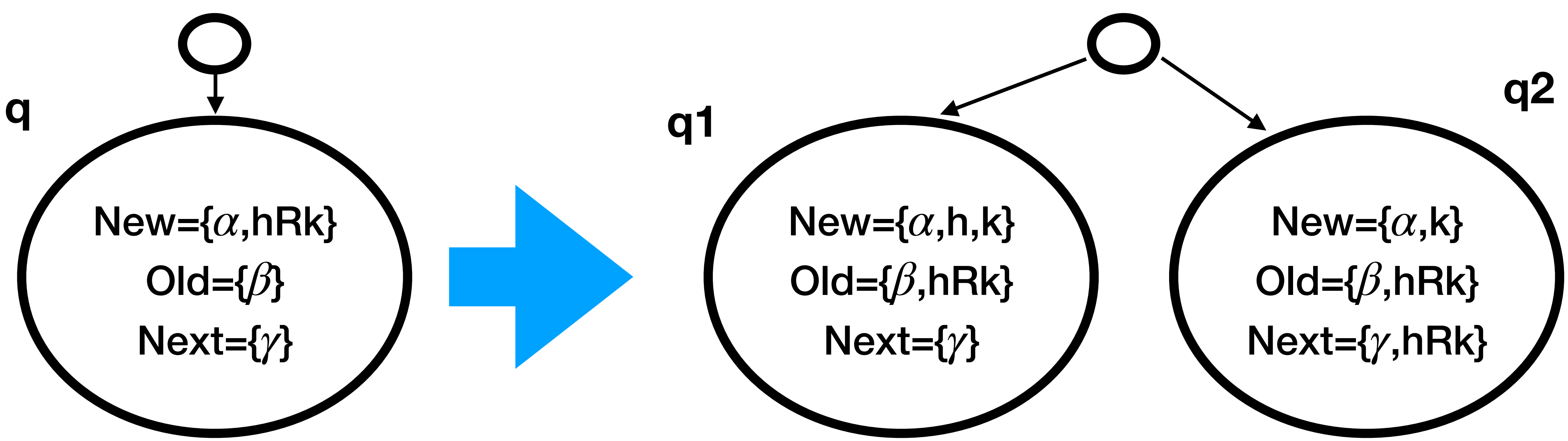
```
ElseIf f≡ h R k
Then
  Create two nodes q1, q2 sucht that
    Incoming(q1):=Incoming(q2):=Incoming(q)
    Old(q1):=Old(q2):=Old(q)∪{h R k}
    New(q1):=New(q)∪{h}∪{k}
    New(q2):=New(q)∪{k}
    Next(q1):=Next(q)
    Next(q2):=Next(q)∪{h R k}

  Return expand(q2, expand(q1, NodeList))
```

Creo dos nodos distintos q1, q2 donde q1 toma "h" y "k", mientras que q2 toma solo "k", pero si vale "k", entonces en el siguiente nodo debe vale hRk

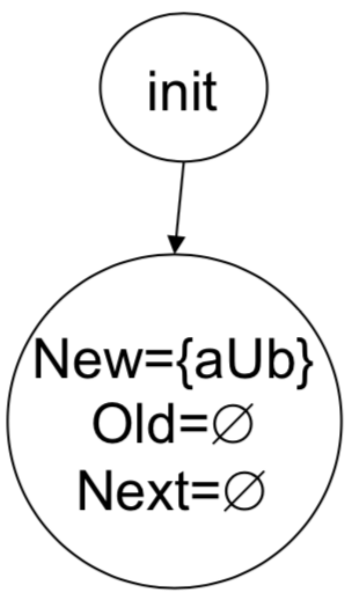
Retorno expandir primero q1 y luego q2

f≡ h R k

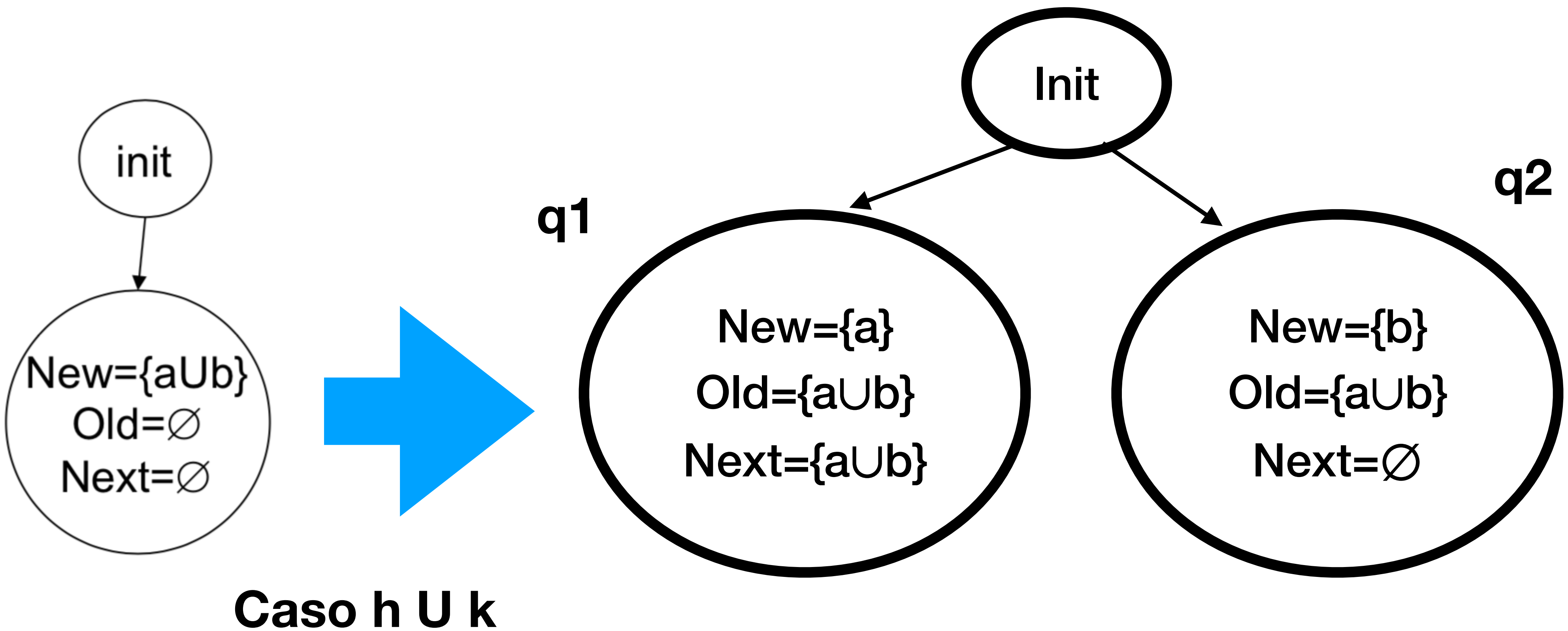
```
ElseIf f≡ h R k
Then
  Create two nodes q1, q2 sucht that
    Incoming(q1):=Incoming(q2):=Incoming(q)
    Old(q1):=Old(q2):=Old(q)∪{h R k}
    New(q1):=New(q)∪{h}∪{k}
    New(q2):=New(q)∪{k}
    Next(q1):=Next(q)
    Next(q2):=Next(q)∪{h R k}

  Return expand(q2, expand(q1, NodeList))
```



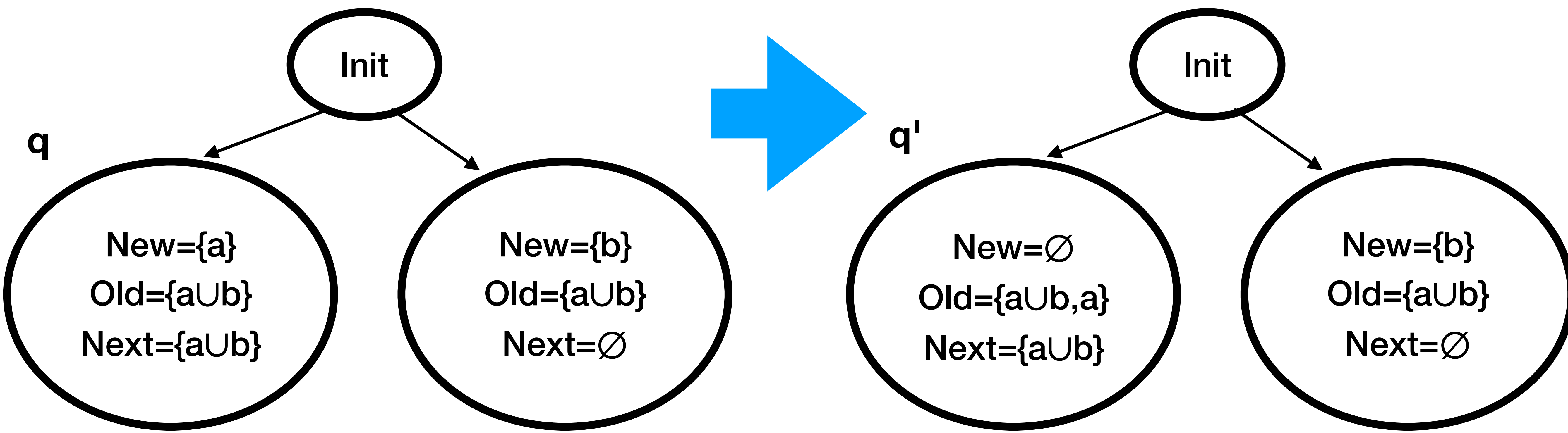
Ejemplo: a U b



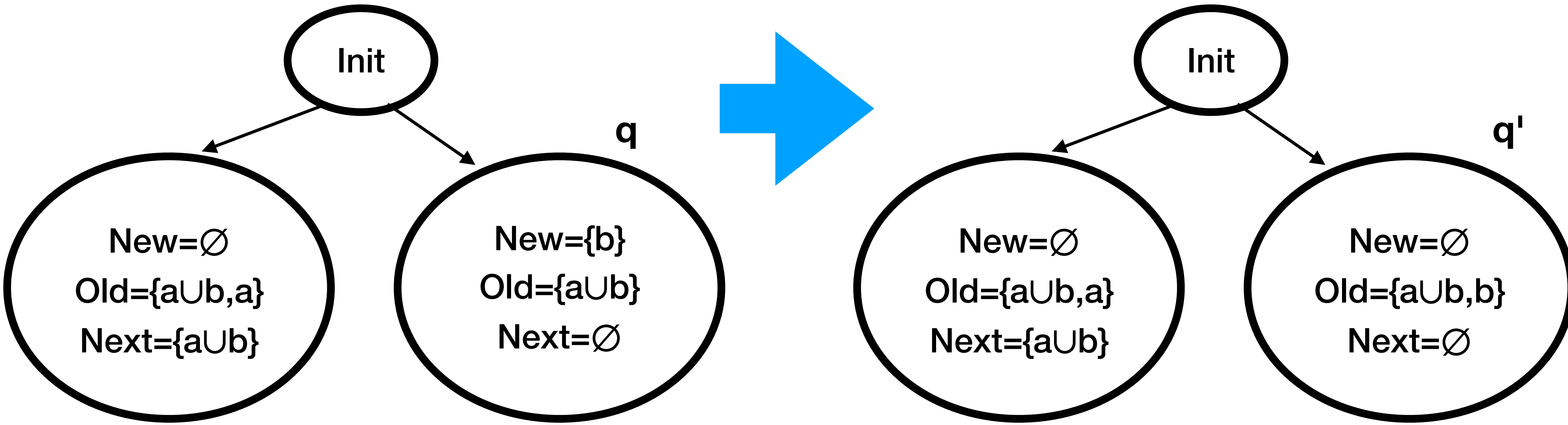
f = h U k


init
New={aUb}
Old=∅
Next=∅
Caso h U k
Init
q1
New={a}
Old={a∪b}
Next={a∪b}
q2
New={b}
Old={a∪b}
Next=∅

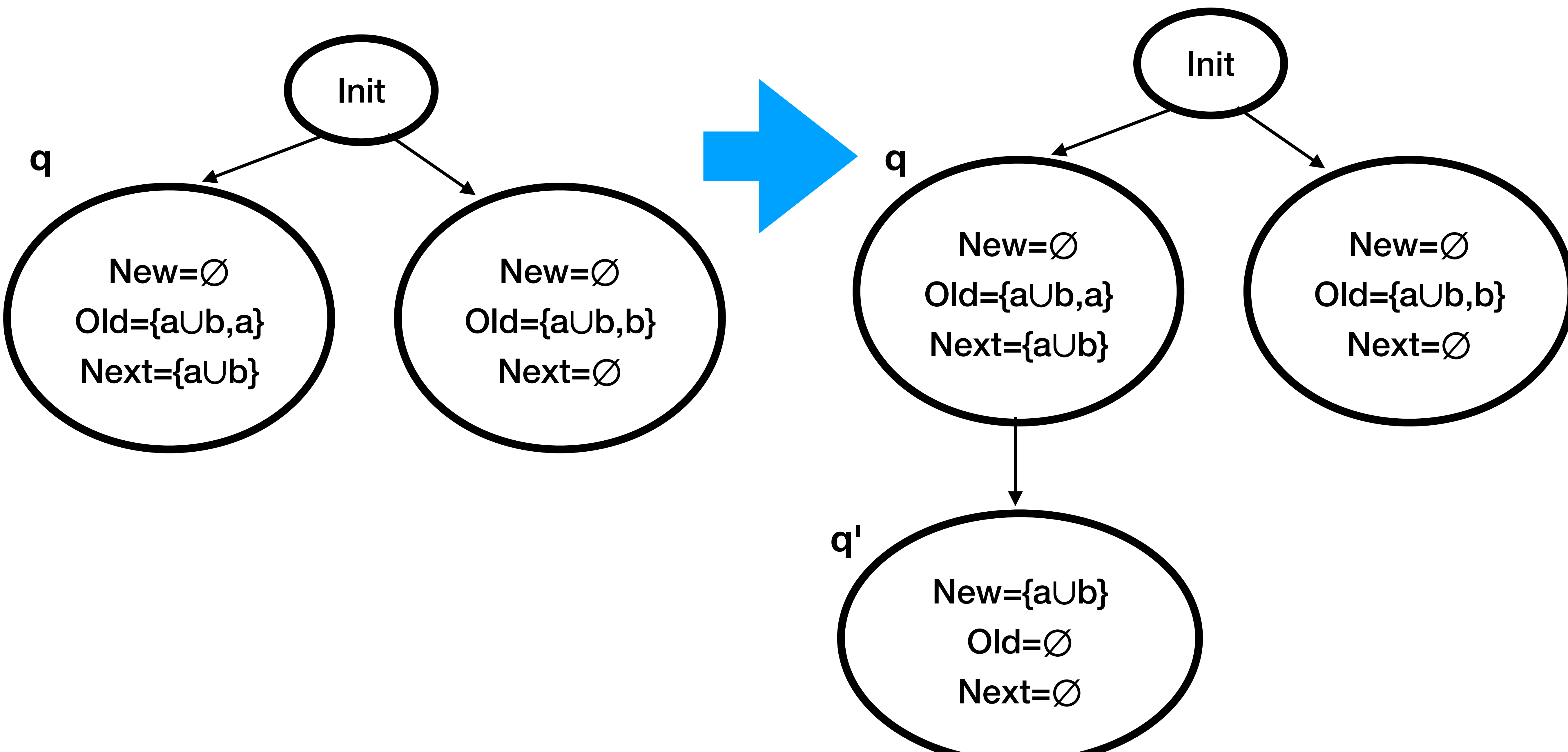
f es proposicion



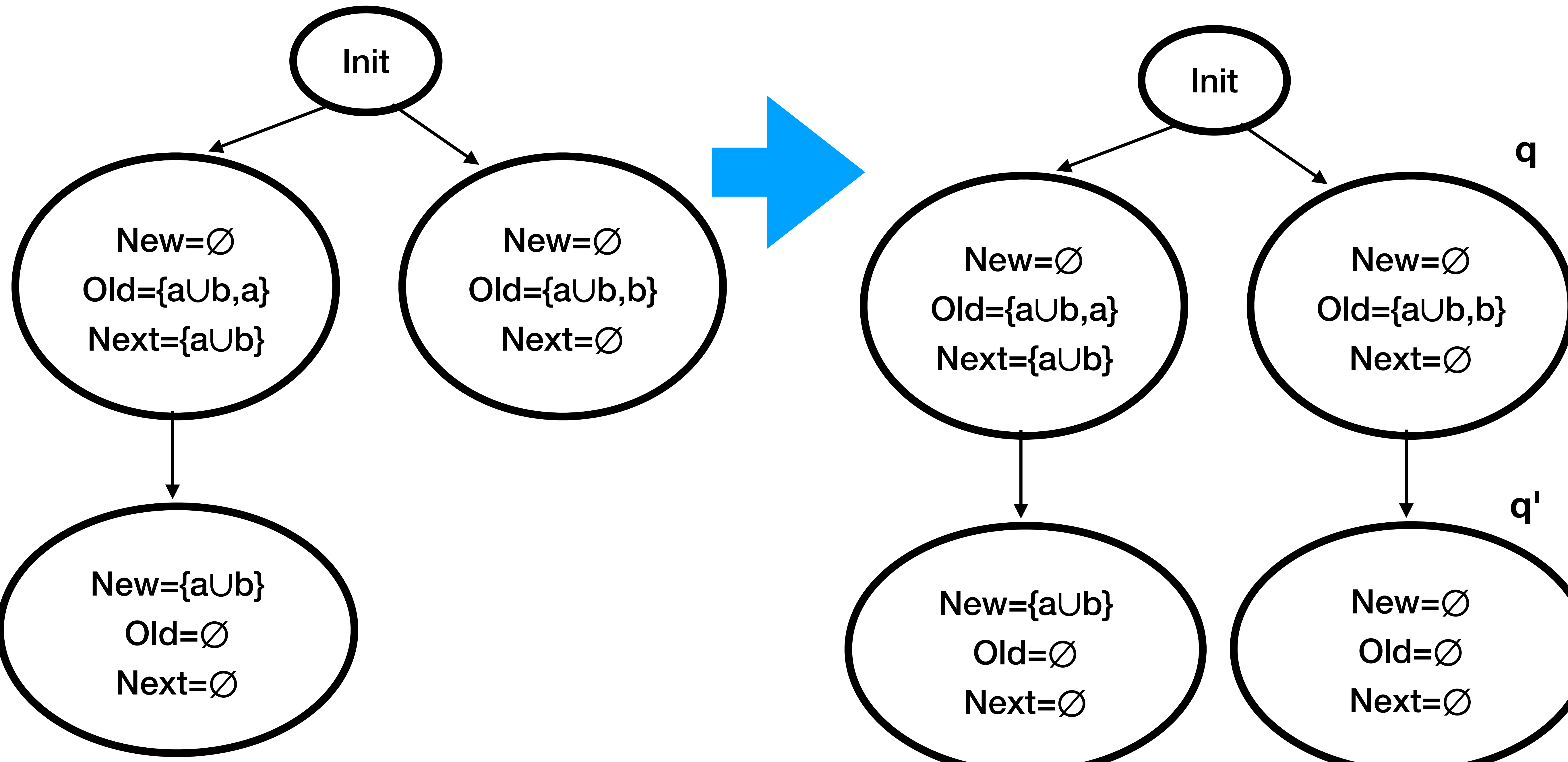
f es proposición



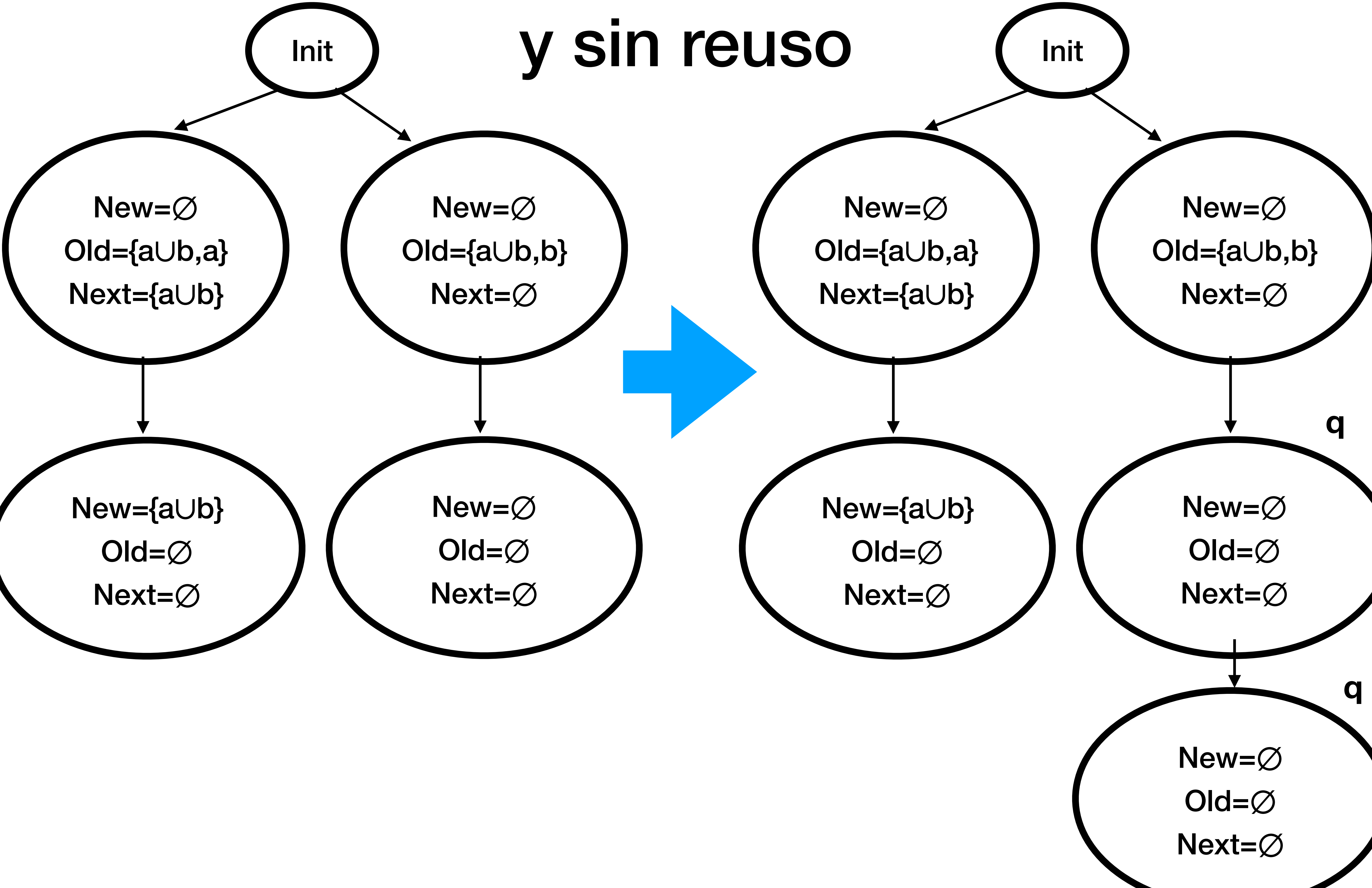
New vacío y sin reuso



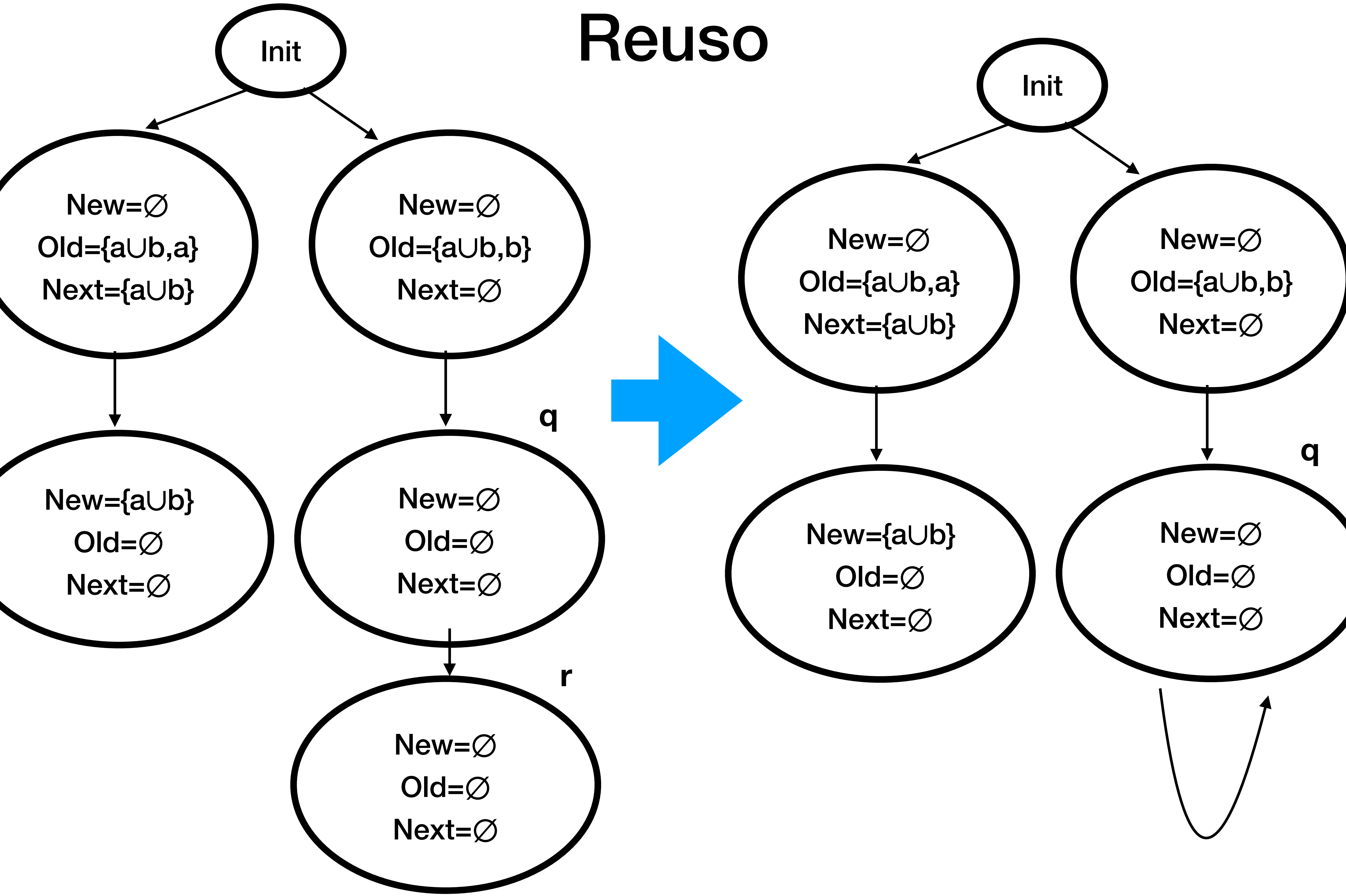
New vacío
y sin reuso

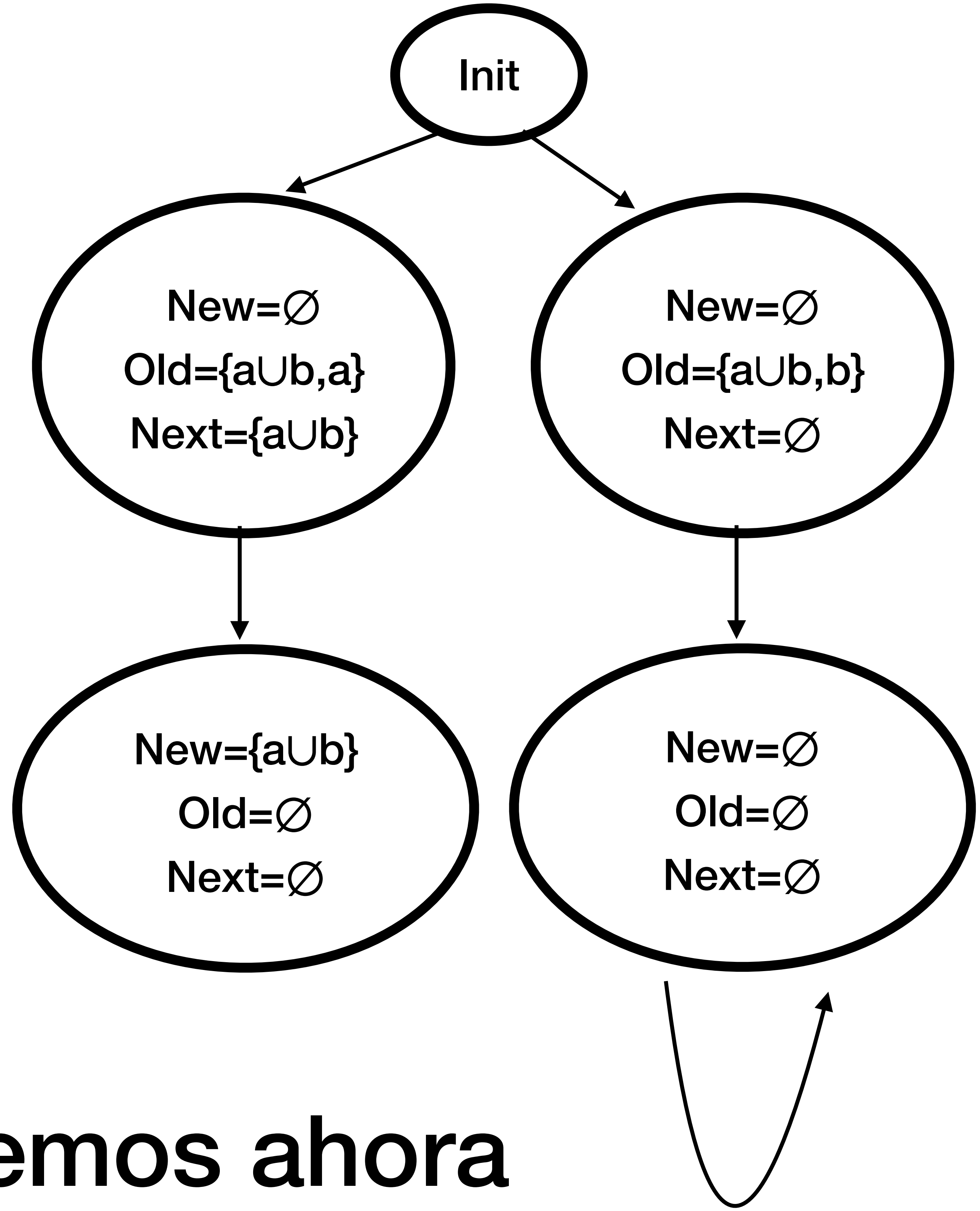


New vacío y sin reuso

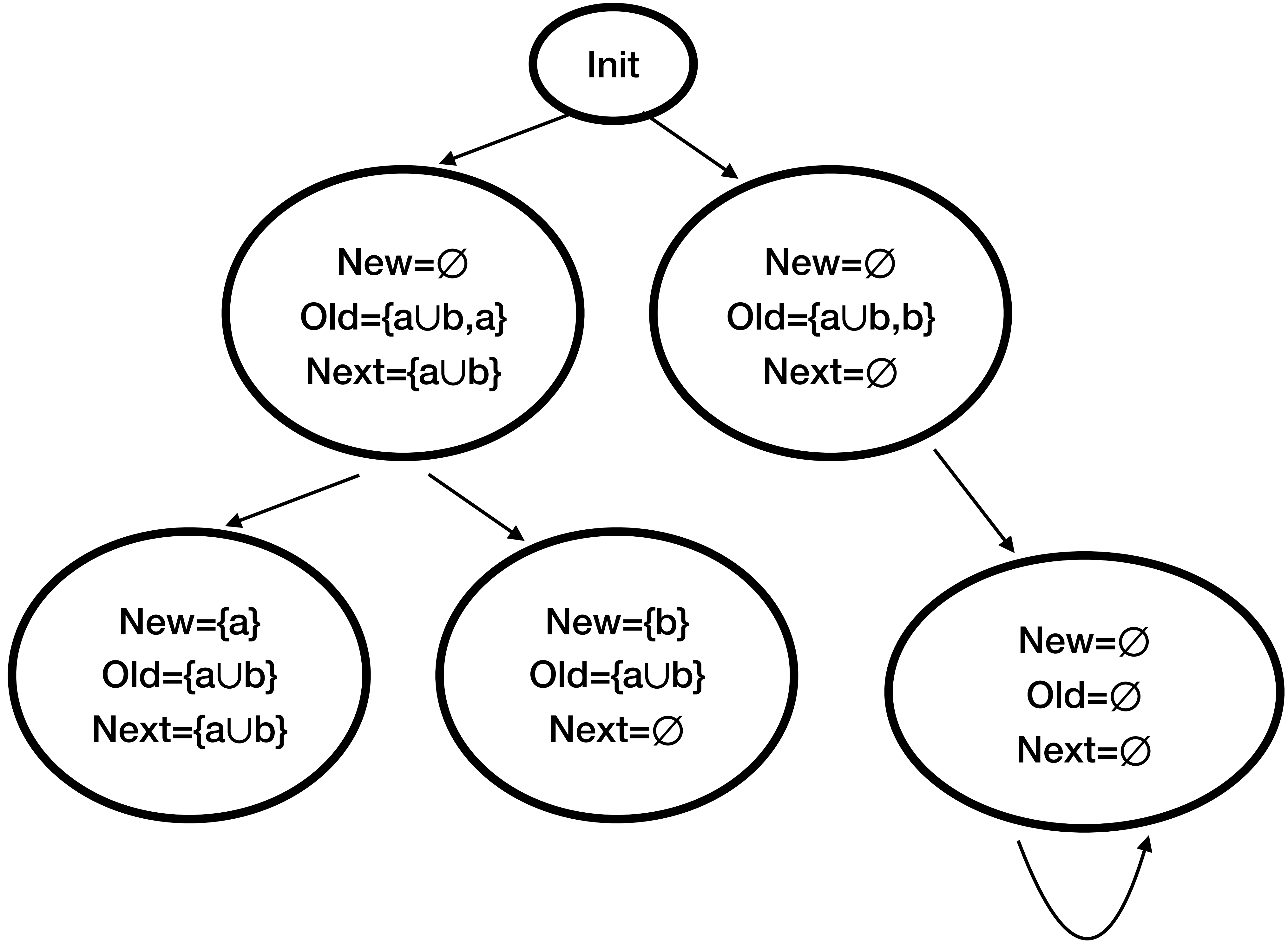


Reuso

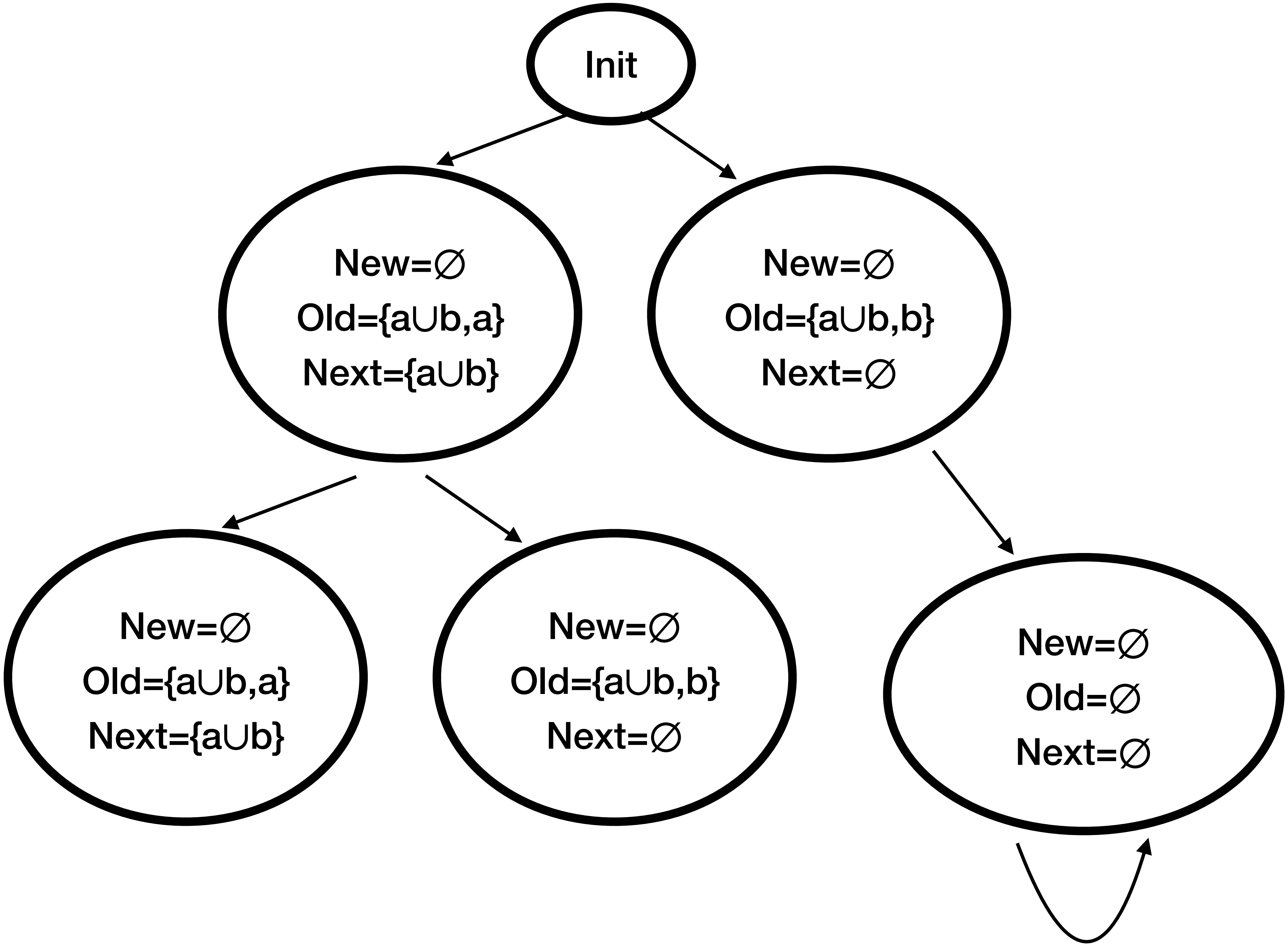

Init
New=∅
Old={a∪b,a}
Next={a∪b}
New=∅
Old={a∪b,b}
Next=∅
New={a∪b}
Old=∅
Next=∅
q
New=∅
Old=∅
Next=∅
r
New=∅
Old=∅
Next=∅
Init
New=∅
Old={a∪b,a}
Next={a∪b}
New=∅
Old={a∪b,b}
Next=∅
New={a∪b}
Old=∅
Next=∅
q
New=∅
Old=∅
Next=∅



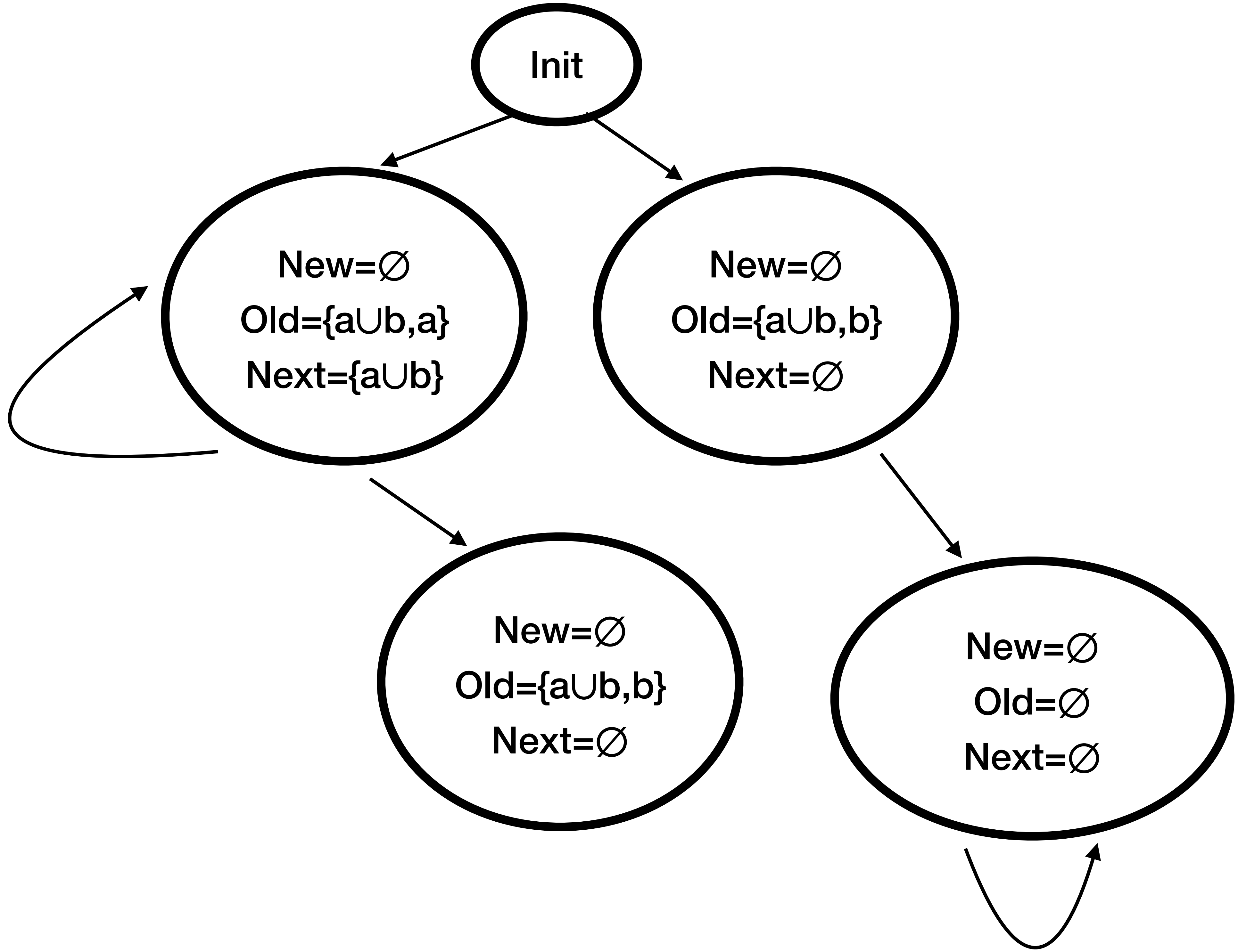
Tomemos ahora
este autómata



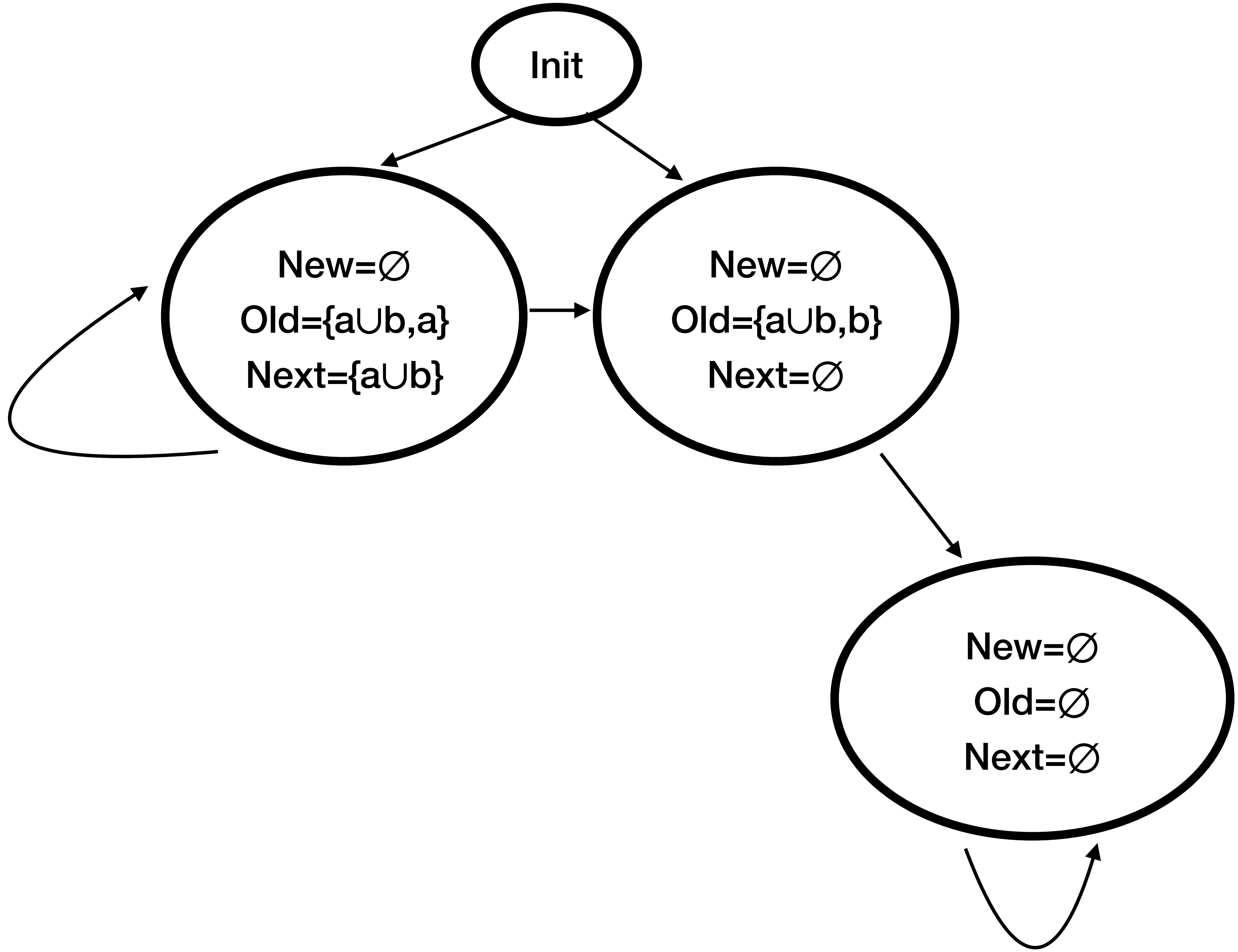
Aplica nuevamente f=hUk



Aplica f es proposición atómica



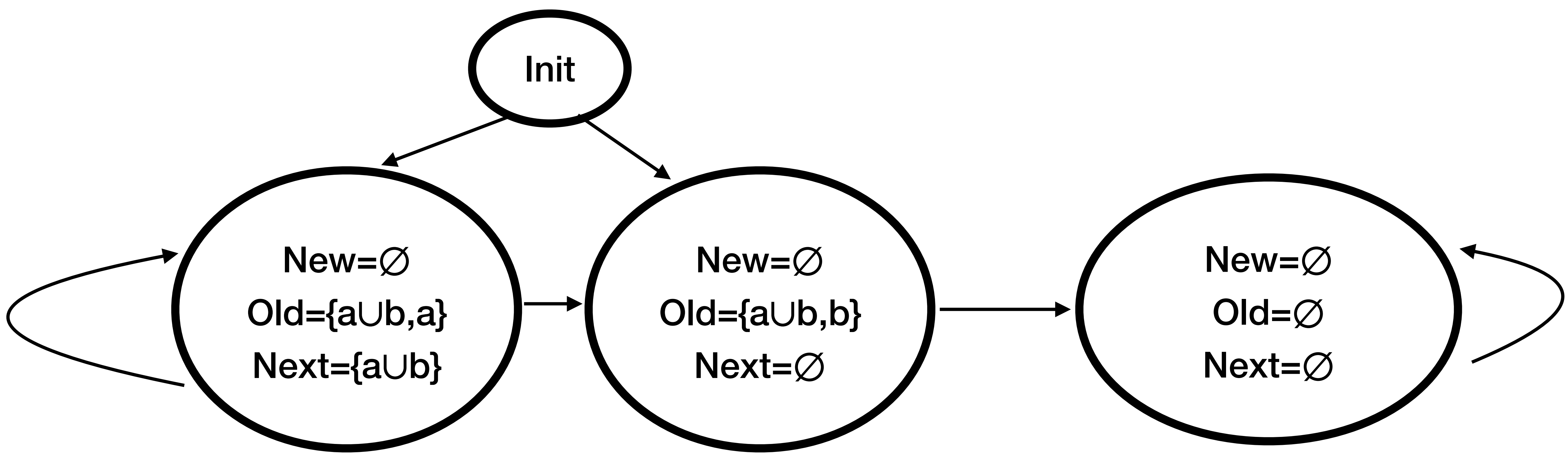
Colapsamos estados equivalentes



Volvemos a colapsar estados equivalentes

Ejemplo: Fórmula aUb

El paso 2 produce el siguiente autómata para el input "aUb"



Recap: LTL2Büchi

[Gerth, Peled, Vardi, Wolper 95]

Input: Formula LTL P

1. Crear un nodo n=<New = {P}, Old =∅, Next=∅, Incoming=∅ >

2. Para cada nodo n con f ∈ new, procesar f creando nuevos nodos. Continuar hasta que no exista f ∈ new en ningún nodo n.

3. Construir un autómata de Büchi generalizado.

4. Traducir el Büchi generalizado en un Büchi común

3 - Construir Autómata de Büchi Generalizado

El autómata de Büchi generalizado resultante es A= $\langle \Sigma, Q, \Delta, Q_0, F \rangle$ donde:

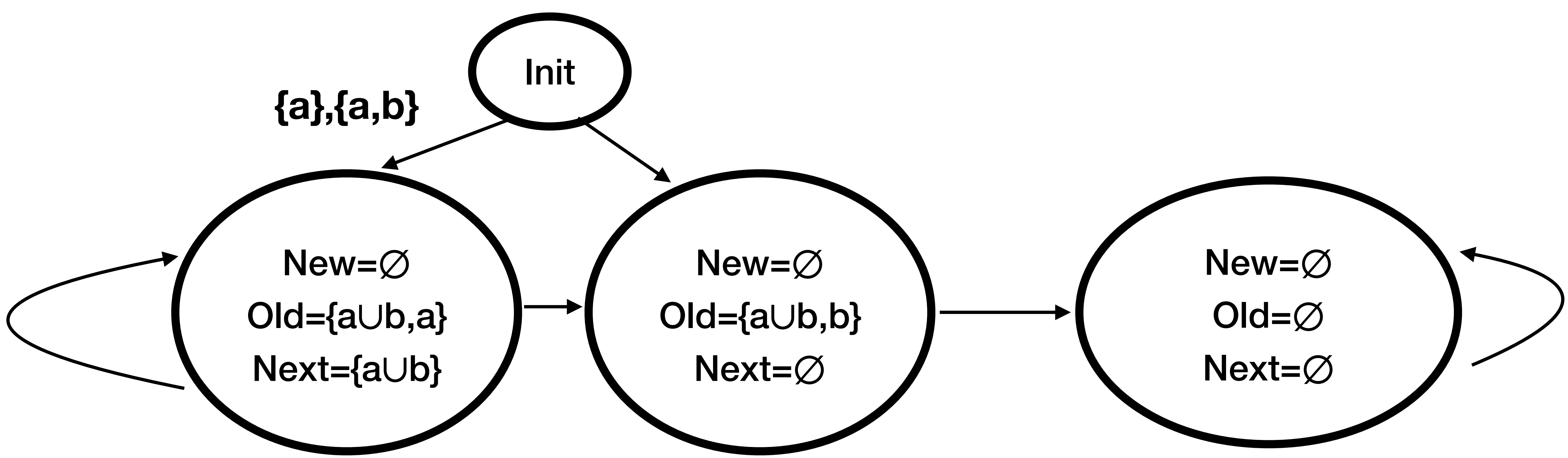
- Σ son subconjuntos de proposiciones de la fórmula LTL
- Q = NodeList U {init}
- Q_0={init}

3 - Construir Autómata de Büchi Generalizado

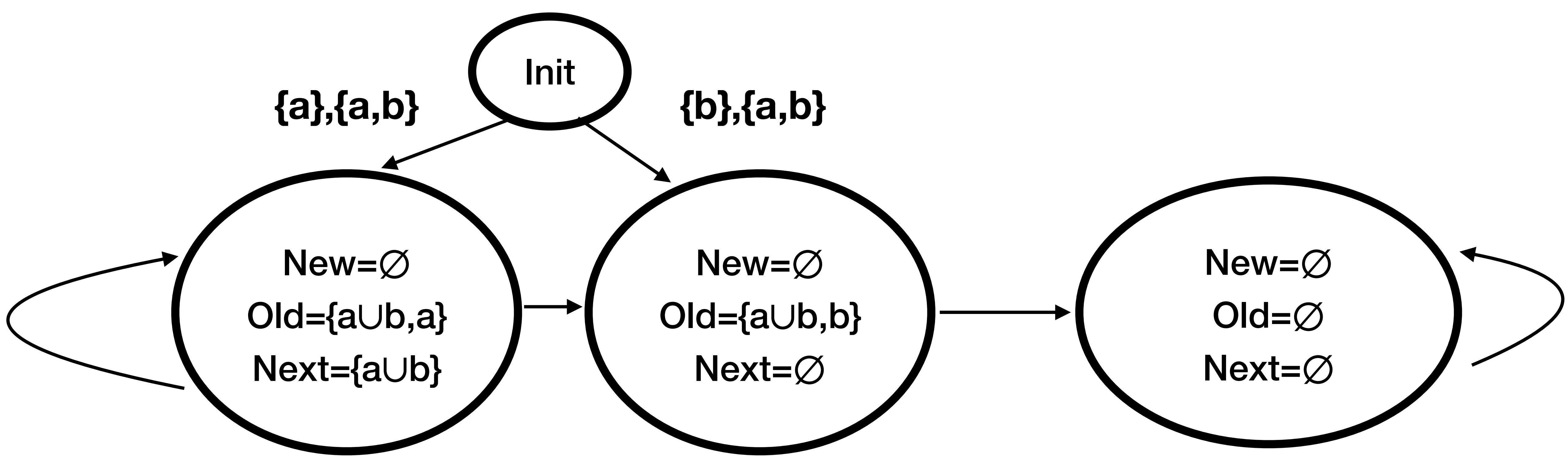
El autómata de Büchi generalizado resultante es A= $\langle \Sigma, Q, \Delta, Q_0, F \rangle$ donde:

- Δ se define del siguiente modo:
  - (q,d,q') $\in$ Δ ssi q $\in$ incoming(q') y d satisface la conjunción de las proposiciones negadas y no negadas que están en Old(q')

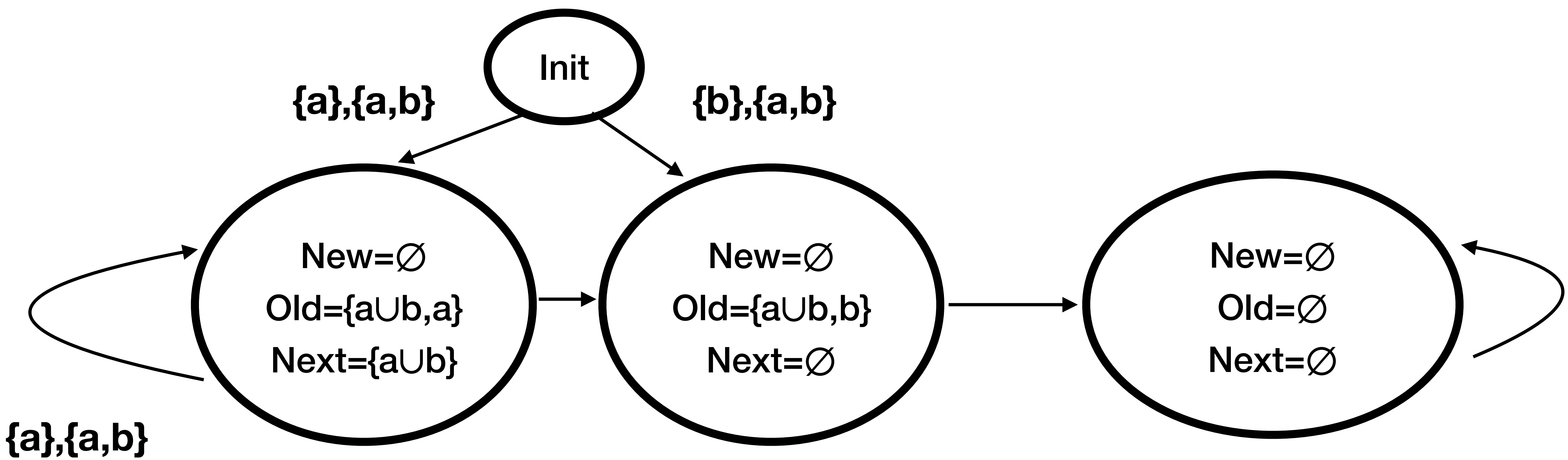
- (q,d,q') ∈ Δ ssi q ∈ incoming(q') y d satisface la conjunción de las proposiciones negadas y no negadas que están en Old(q')



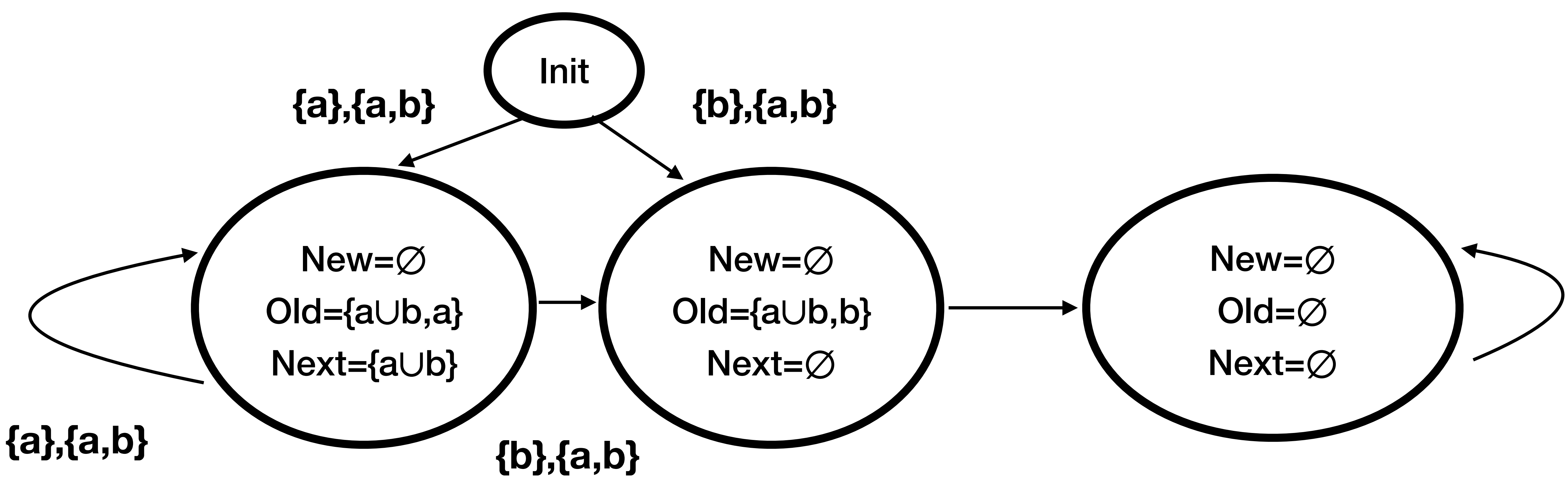
- (q,d,q') ∈ Δ ssi q ∈ incoming(q') y d satisface la conjunción de las proposiciones negadas y no negadas que están en Old(q')



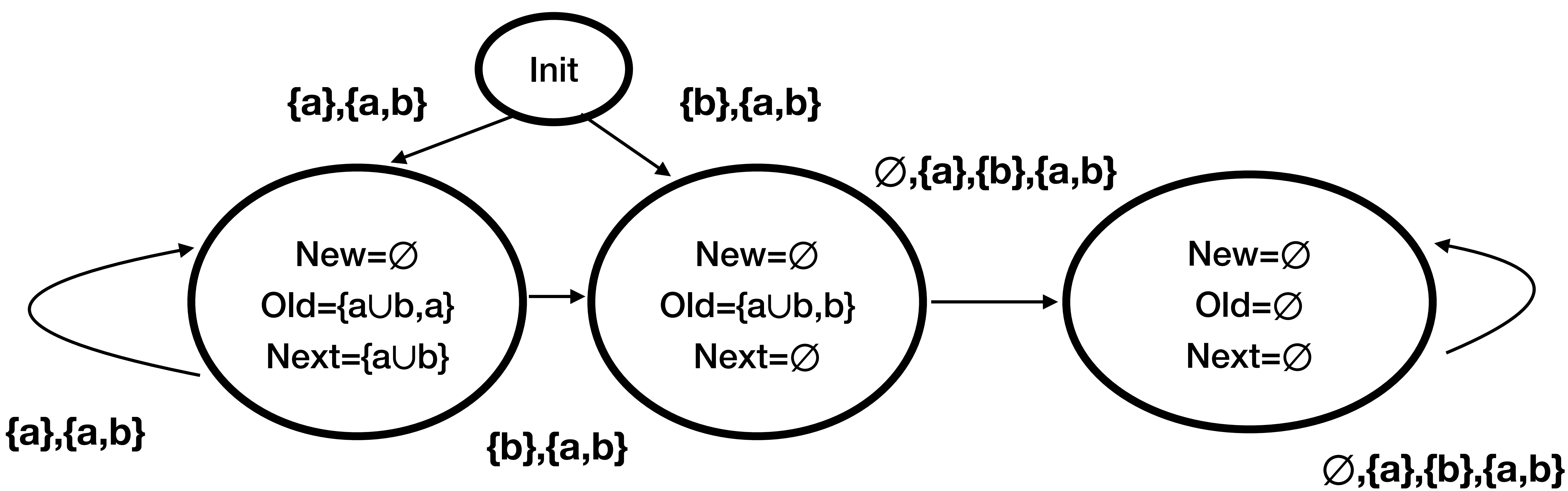
- (q,d,q') ∈ Δ ssi q ∈ incoming(q') y d satisface la conjunción de las proposiciones negadas y no negadas que están en Old(q')



- (q,d,q') ∈ Δ ssi q ∈ incoming(q') y d satisface la conjunción de las proposiciones negadas y no negadas que están en Old(q')



- (q,d,q') ∈ Δ ssi q ∈ incoming(q') y d satisface la conjunción de las proposiciones negadas y no negadas que están en Old(q')

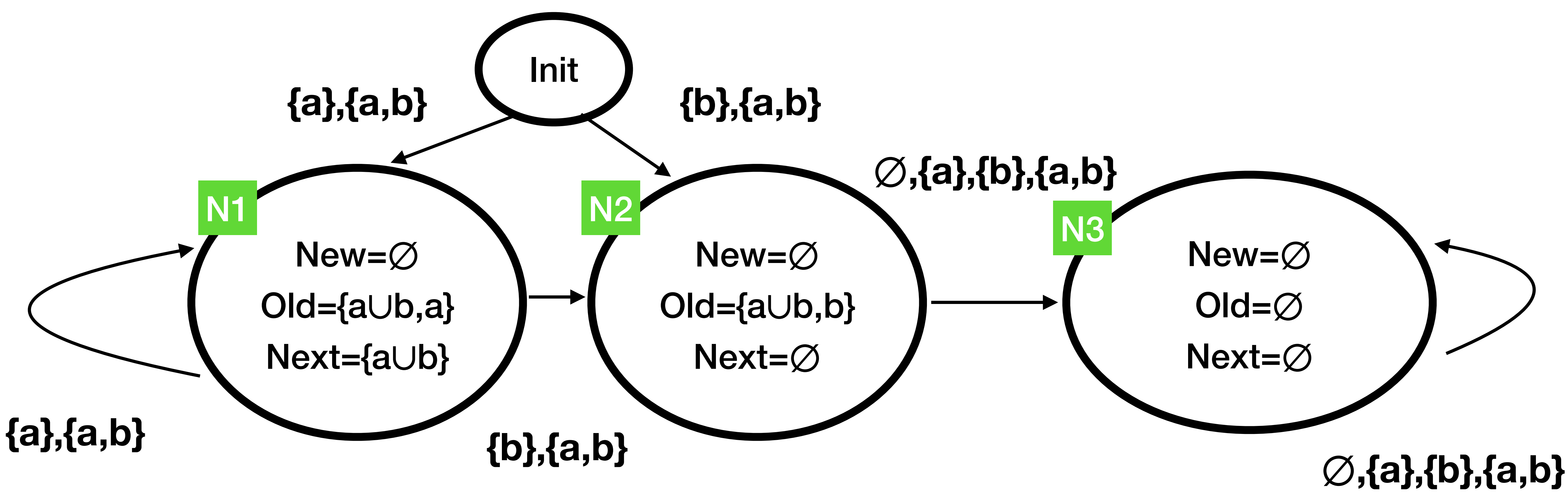


3 - Construir Autómata de Büchi Generalizado

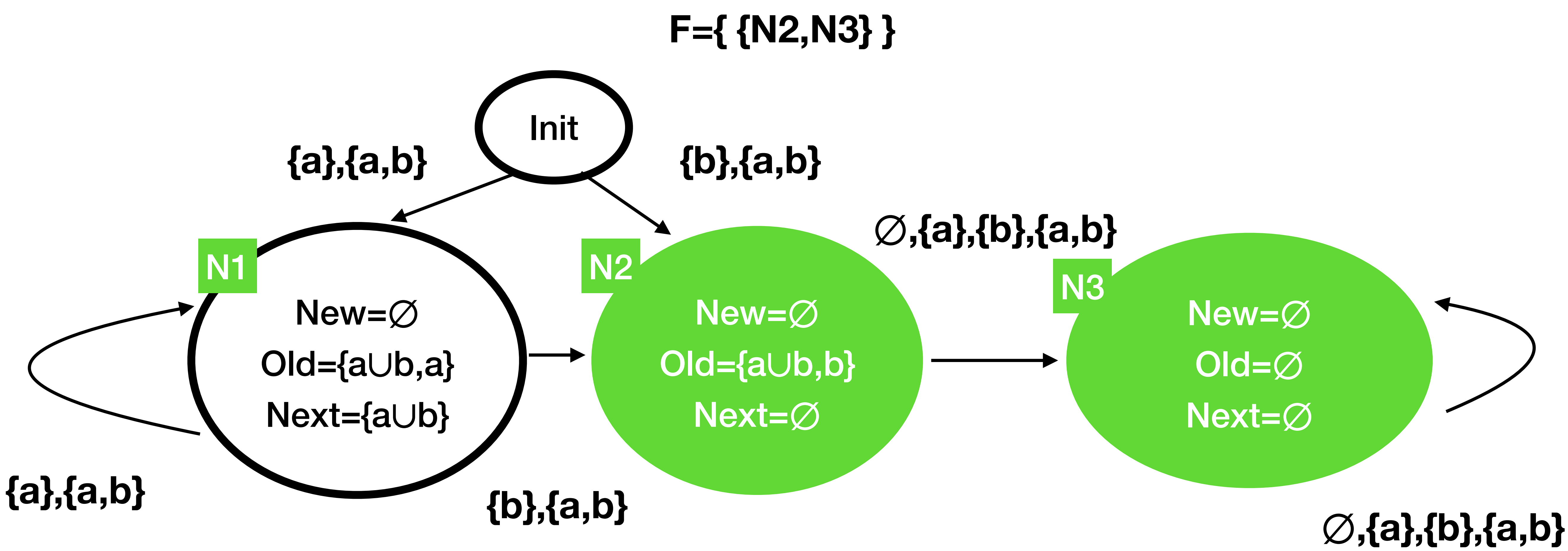
El autómata de Büchi generalizado resultante es A= $\langle \Sigma, Q, \Delta, Q_0, F \rangle$ donde:

- F $\subseteq 2^Q$ i.e., F={$F_1, F_2, \ldots$} se define del siguiente modo:
  - Para cada sub-fórmula h U k existe un estado de aceptación F_i que contiene todos los estados q tales que o bien "k" $\in$ Old(q) o bien "h U k" $\notin$ Old(q)
  - Si no hay sub-fórmulas de la forma "h U k" , entonces F = {Q}

- Para cada sub-fórmula h U k existe un estado de aceptación F_i que contiene todos los estados q tales que o bien "k" $\in$ Old(q) o bien "h U k" $\notin$ Old(q)

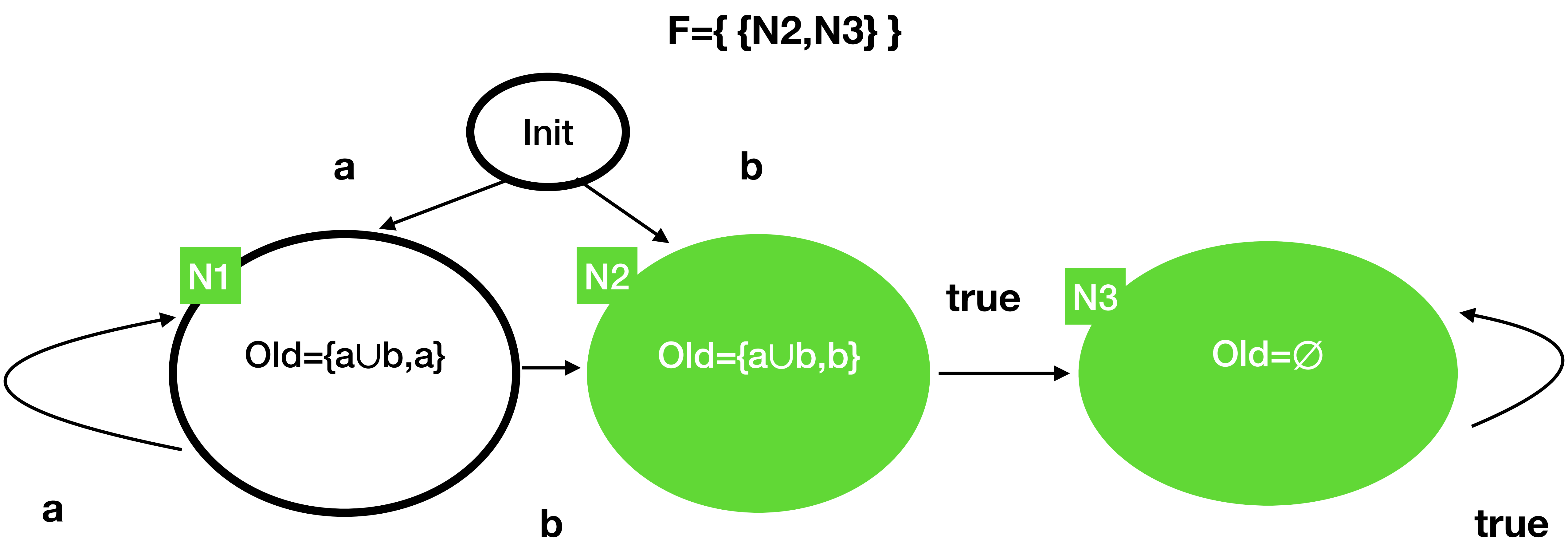


- Para cada sub-fórmula h U k existe un estado de aceptación F_i que contiene todos los estados q tales que o bien "k" ∈ Old(q) o bien "h U k" ∉ Old(q)



- Finalmente, podemos cambiar los conjuntos de valuaciones en una transición por una formula proposicional que los caracteriza

- Este es el autómata de Büchi generalizado que acepta las trazas que satisfacen la fórmula LTL "a U b"



Tiempo de Ejecución

- La complejidad del algoritmo es **exponencial** con respecto al tamaño de la formula

Recap: LTL2Büchi

[Gerth, Peled, Vardi, Wolper 95]

Input: Formula LTL P

1. Crear un nodo n=<New = {P}, Old =∅, Next=∅, Incoming=∅ >

2. Para cada nodo n con f ∊ new, procesar f creando nuevos nodos. Continuar hasta que no exista f ∊ new en ningún nodo n.

3. Construir un autómata de Büchi generalizado.

4. Traducir el Büchi generalizado en un Büchi común

4- Büchi Generalizado a Büchi

- Sea GA=$\langle \Sigma, Q, \Delta, Q_0, \{F_1, \ldots, F_k\} \rangle$ un autómata de Büchi *generalizado*, entonces podemos construir un autómata de Büchi A del siguiente modo:

- A=$\langle \Sigma, Q \times \{1,\ldots,k\}, \Delta', Q_0, F_1 \times \{1\} \rangle$ donde

- ((q,x), a, (q',y)) $\in \Delta'$ sii

  - (q,a,q') $\in \Delta$

  - si q$\in F_i$ $\wedge$ x=i<k entonces y=i+1

  - si q$\in F_k$ $\wedge$ x=i entonces y=1

  - no existe i tal que q$\in F_i$ y además x=y

Lema

- Sea GA=$\langle \Sigma, Q, \Delta, Q_0, \{F_1\} \rangle$ un autómata de Büchi *generalizado* con un único conjunto de aceptación,

- Entonces podemos construir el autómata A= $\langle \Sigma, Q, \Delta, Q_0, F_1 \rangle$ es un autómata de Büchi (no generalizado) tal que *L(GA)=L(A)*

- Demostración (Ejercicio)

- Este es el autómata de Büchi no-generalizado que acepta las trazas que satisfacen la fórmula LTL "a U b"